THM | CAMPUS GIESSEN | MNI
TECHNISCHE HOCHSCHULE MITTELHESSEN
Mathematik, Naturwissenschaften und Informatik

Masterarbeit

Entwicklung eines generischen, quantensicheren Watchdog-Timer-Protokolls

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

vorgelegt dem

Fachbereich Mathematik, Naturwissenschaften und Informatik an der
der Technischen Hochschule Mittelhessen

Tanja Gutsche

am 30. Januar 2023

Referent:	Prof. Dr.-Ing. André Rein
Korreferent:	Prof. Dr. Hagen Lauer

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Gießen, 30. Januar 2023

_____

Unterschrift

Danksagung

Diese Masterarbeit wurde zwischen August 2022 und Januar 2023 in Kooperation mit dem Fraunhofer-Institut für Sichere Informationstechnologie (SIT) mit Sitz in Darmstadt verfasst. Daher möchte ich mich zuallererst für die Ermöglichung des Verfassens meiner Abschlussarbeit am Fraunhofer SIT in Darmstadt bedanken. Ich hoffe, dass die Erkenntnisse aus dieser Masterarbeit eine Unterstützung bei der Umsetzung von Projekten des Fraunhofer SIT darstellen. Hierbei möchte ich einen besonderen Dank an Herrn Michael Eckel (Fraunhofer SIT) aussprechen, der mich während der Bearbeitung der Masterarbeit betreute und mir bei Fragen mit Rat und Tat zur Seite stand. Ein weiterer Dank gilt Herrn Prof. Dr. Hagen Lauer, der mich sowohl am Fraunhofer SIT als auch als Referent vonseiten der THM betreute und mich bei der Themenfindung der Masterarbeit unterstützt hat, sowie mich mit weiteren Personen bekannt gemacht hat, die ähnliche Themen bearbeiten. Für meine anfänglichen Schritte am Fraunhofer SIT bedanke ich mich außerdem bei Herrn Richard Petri und Herrn Norman Lahr, die mir einen Einblick in die Post-Quantum-Kryptographie im Zusammenhang mit eingebetteten Systemen ermöglicht haben. Dadurch habe ich ein Teilgebiet des Themas dieser Masterarbeit kennengelernt.

Vonseiten meiner Hochschule, der Technischen Hochschule Mittelhessen am Standort Gießen, gilt ein großer Dank Herrn Prof. Dr.-Ing. André Rein. Er war mein Mentor während meiner Zeit im Masterstudium und hat mich bei der Organisation, von der Bewerbung beim Fraunhofer SIT bis hin zur Betreuung der Masterarbeit stark unterstützt. Des Weiteren möchte ich mich bei ihm für viele interessante Gespräche bedanken.

Ein besonderer Dank geht an meinen Freund, für seine Begleitung durch alle Höhen und Tiefen während der Bearbeitung der Masterarbeit. Außerdem möchte ich mich bei ihm für das Korrekturlesen meiner Masterarbeit bedanken.

Ein großer Dank geht außerdem an meine Eltern, meine Familie und meine Freunde, die mich immer unterstützen und für mich da sind.

Abstract

This work aims to design and implement a generic, quantum-safe watchdog timer protocol using various quantum-safe digital signature algorithms. The protocol is to be applied in the field of the Internet of Things.

After answering the research question of how a classical watchdog timer protocol is implemented, replacing the classical protocol with a quantum-safe watchdog timer protocol follows. Subsequently, the research question arises as to how high the performance difference between the two versions of the protocol is, concerning the exchanged functions and particular processes of the protocol.

A combination of methods is used to answer the research questions. First, a concept of the watchdog timer protocol is developed, considering possible scenarios, which is successfully verified by a proof-of-concept implementation in Python. Then, measurements to compare different algorithms are performed and quantitatively evaluated by statistical methods. This thesis uses an inductive research approach, as the results are used to establish theories at the end to answer the research questions.

The replacement of quantum-safe signature algorithms in the watchdog timer protocol is successfully implemented. The implementation and evaluation of measurements of CPU cycles show that the performance of tested quantum-safe algorithms is better or similar compared to tested classical algorithms. A quantum-safe algorithm is recommended for use with the watchdog timer protocol in terms of performance.

Zusammenfassung

Das Ziel dieser Arbeit ist die Konzeption und Implementierung eines generischen, quantensicheren Watchdog-Timer-Protokolls unter Verwendung verschiedener quantensicherer digitaler Signaturalgorithmen. Das Protokoll soll im Bereich des Internet of Things Anwendung finden.

Nach der Beantwortung der Forschungsfrage, wie ein klassisches Watchdog-Timer-Protokoll umgesetzt wird, folgt die Ersetzung des klassischen Protokolls durch ein quantensicheres Watchdog-Timer-Protokoll. Anschließend stellt sich die Forschungsfrage, wie hoch der Performance-Unterschied der beiden Versionen des Protokolls in Bezug auf die ausgetauschten Funktionen und bestimmte Abläufe im Protokoll ist.

Um die Forschungsfragen zu beantworten, wird eine Kombination aus verschiedenen Methoden angewandt. Zunächst wird ein Konzept des Watchdog-Timer-Protokolls unter der Berücksichtigung möglicher Szenarien entwickelt, das durch eine Proof-of-Concept-Implementierung in Python erfolgreich verifiziert wird. Anschließend werden Messungen zum Vergleichen verschiedener Algorithmen durchgeführt und quantitativ durch statistische Methoden ausgewertet. Es handelt sich um einen induktiven Forschungsansatz, da mit den Ergebnissen am Ende Theorien zur Beantwortung der Forschungsfragen aufgestellt werden.

Die Ersetzung quantensicherer Signaturalgorithmen im Watchdog-Timer-Protokoll wurde erfolgreich umgesetzt. Die Durchführung und Auswertung von Messungen der CPU-Zyklen zeigt, dass die Performance von getesteten quantensicheren im Vergleich zu getesteten klassischen Algorithmen besser bzw. annähernd gleich ist. Es wird ein quantensicherer Algorithmus für die Verwendung des Watchdog-Timer-Protokolls in Bezug auf Performance empfohlen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AES	Advanced Encryption Standard
API	Application Programming Interface
AWDT	Authenticated Watchdog Timer
BKA	Bundeskriminalamt
BSI	Bundesamt für Sicherheit in der Informationstechnik
CMAC	Cipher-based Message Authentication Code
CPU	Central Processor Unit
CRYSTALS	**CRY**ptographic **S**ui**T**e for **A**lgebaric **L**attice**S**
DHKE	Diffie-Hellman Key Exchange
DICE	Device Identifier Composition Engine
DICE++	Erweiterung für DICE
DLP	Diskretes Logarithmus-Problem
ECC	Elliptische Kurven Kryptografie
ECDH	Elliptic Curve Diffie-Hellman
ECDLP	Elliptic Curve Discrete Logarithm Problem
ECDSA	Elliptic Curve Digital Signature Algorithm
EU	Europäische Union
FALCON	**Fa**st-Fourier **L**attice-based **C**ompact Signatures **o**ver **N**TRU
HACL	High-Assurance Cryptographic Library
HMAC	Hash-Based Message Authentication Code
HRNG	Hardware Random Number Generator
IoT	Internet of Things
IT	Informationstechnologie
JSON	JavaScript Object Notation
KEM	Key Encapsulation Mechanism

LWE	Learning With Errors
MAC	Message Authentication Code
NASA	National Aeronautics and Space Administration
NIST	National Institute of Standards and Technology
OQS	Open Quantum Safe
PIN	Personal Identification Number
PQC	Post-Quantum-Kryptographie
RAM	Random Access Memory
RSA	Rivest-Shamir-Adleman-Algorithmus
SAFEcrypto	Secure Architectures of Future Emerging Cryptography
SIDH	Supersingular Isogeny Diffie–Hellman Key Exchange
SIEM	Security Information and Event Management
SIT	Sichere Informationstechnologie
SHA	Secure Hash Algorithm
SSL	Secure Socket Layer
SVP	Shortest Vector Problem
TAN	Transaktionsnummer
TCB	Trusted Computing Base
TCG	Trusted Computing Group
TCP	Transmission Control Protocol
TEE	Trusted Execution Environment
TLS	Transport Layer Security
TPM	Trusted Platform Module
TRL	Technology Readiness Level
UML	Unified Modeling Language
WLAN	Wireless Local Area Network
XMSS	Extended Merkle Signature Scheme

1. Einleitung

Das Internet of Things (IoT) (zu Deutsch: Internet der Dinge) ist ein breites Feld, bestehend aus verschiedenen vernetzten Objekten. Nach Dorsemaine *et al.* [1] besteht das IoT aus einer „Gruppe von Infrastrukturen, die vernetzte Objekte miteinander verbinden und deren Verwaltung, Data Mining und den Zugriff auf die von ihnen erzeugten Daten ermöglichen."

Aktuell verbreitet sich das Internet of Things immer weiter. Es besteht aus vielen Geräten, die eine bestimmte Aufgabe erfüllen sollen und mit einem Server über ein meist kabelloses Netzwerk verbunden sind [2]. Es gibt verschiedene Klassen an IoT-Geräten, deren Standorte mobil oder fest sind und solche die ununterbrochen mit dem Server kommunizieren oder nur periodisch Daten austauschen. Manche IoT-Geräte müssen durchgängig am Stromnetz angeschlossen sein und andere nicht [1]. Beispiele sind Smartcards [1], Monitoring von Aufzügen oder der Luftqualität, sowie Verkehrskontrollsysteme [3].

Zum IoT zählen u.a. kritische Infrastrukturen, wie z.B. intelligente Energiesysteme, die auch in Zukunft sicher funktionieren sollen. Des Weiteren sollen kritische Infrastrukturen nach einem Angriff schnell wiederhergestellt werden und dürfen durch Angriffe nicht funktionsuntüchtig werden. Viele dieser Geräte sind ressourcenbeschränkt und können nur wenig Speicherplatz sowie Rechenleistung aufweisen und benötigen dementsprechend mehr Rechenzeit [4]. Generell kommt die Sicherheit solcher weit verbreiteter IoT-Geräte zu kurz, was viele Angriffe in der Vergangenheit zeigen, wie z.B. von Botnetzen [5] [6]. Um die Sicherheit von IoT-Geräten bereitzustellen, müssten einige kostenintensive Anforderungen umgesetzt werden, die im aktuellen Preisrahmen der Geräte nicht umsetzbar sind [5].

Um die Sicherheit von IoT-Geräten bereitzustellen, können Protokolle genutzt werden, die Algorithmen der klassischen Kryptografie zum Verschlüsseln von Nachrichten und Authentifizieren von Kommunikationspartnern verwenden. Laut Bundesamt für Sicherheit in der Informationstechnik (BSI) [7] basieren die asymmetrischen kryptografischen Algorithmen auf mathematischen Problemen wie Faktorisierung und dem Diskretes Logarithmus-Problem (DLP), die mit klassischen Computern nicht in polynomieller Zeit lösbar sind. Shor [8] fand jedoch im Jahr 1994 heraus, dass mit quantenmechanischen Computern die genannten mathematischen Probleme in polynomieller Zeit gelöst werden können und somit die asymmetrischen Algorithmen gebrochen werden können.

Nicht alle heutzutage verwendeten Verschlüsselungsalgorithmen können von Quantencomputern gebrochen werden. Grover's Algorithmus sorgt dafür, dass bei Suchproblemen in der symmetrischen Kryptografie die Suche nach einem geheimen Schlüssel quadratisch beschleunigt werden kann [7]. Zum Finden des geheimen Schlüssels für AES-128 wird der Zeitaufwand von 2^{128} auf 2^{64} reduziert. Um die Sicherheit eines Algorithmus zu erhöhen, kann

die Schlüssellänge erhöht werden. Laut BSI [7] gibt es noch weitere Quantenalgorithmen, die klassische Probleme schneller lösen können. In Zukunft könnten Quantenalgorithmen entwickelt werden, die auch die symmetrische Kryptografie brechen können [9].

1.1. Problembeschreibung und Motivation

In den letzten Jahren sind die erfassten Cyberkriminalitätsfälle in Deutschland gestiegen. Im Jahr 2021 gab es 124.137 Fälle, von denen ca. ein Drittel aufgeklärt wurde [10]. Laut dem Bundeskriminalamt (BKA) war 2021 geprägt von Ransomware-Angriffen auf kritische Infrastrukturen, öffentliche Verwaltungen und internationale Lieferketten [10]. Zu kritischen Infrastrukturen zählen u.a. Windkraftanlagen. Es ist wichtig, dass diese trotz Kompromittierung ihre Aufgabe weiterhin einwandfrei erledigen. Um das System bzw. das Gerät, das die Windkraftanlage steuert, schnell wieder in einen sicheren, vertrauenswürdigen Zustand zu bringen, kann mithilfe eines Watchdog-Timers der Reboot eines Geräts kontrolliert hervorgerufen werden. Die Häufigkeit des Resets hängt vom Anwendungsfall ab. Durch diese Zustandswiederherstellung ist ein Angriff wirtschaftlich nicht rentabel, da ein Reboot ggf. schneller erfolgt als der Angreifer benötigt, um das System erneut zu kompromittieren.

Bei großen Rechenzentren gibt es die Möglichkeit, dass die Administratoren Software-Updates - z.B. zur Wiederherstellung - auf allen Servern installieren können, unabhängig vom Zustand, in dem sich die Server jeweils befinden [11]. Diese Möglichkeit besteht bei IoT-Geräten nicht, da diese u.a. beschränkte Ressourcen im Vergleich zu Servern besitzen [3]. Daher muss eine Alternative für den IoT-Bereich gefunden werden [3].

Manche IoT-Geräte haben eine Trusted Execution Environment (TEE) im Prozessor integriert, womit unsicherer Code - wie der zur Kommunikation zwischen Gerät und Server - von der sogenannten Trusted Computing Base (TCB) und dem zu schützenden Code isoliert werden kann [2]. Dadurch können z.B. Seitenkanalangriffe vermieden werden und Sicherheitshardware, die für sogenannte cyber-resiliente Architekturen benötigt wird, kann mit einer TEE emuliert werden [2]. Huber *et al.* [2] gehen davon aus, dass TEEs in Low-End-Mikroprozessoren in Zukunft weiter verbreitet sind. Daher wird im Folgenden davon ausgegangen, dass in den Mikroprozessoren eine TEE integriert ist und genutzt werden kann.

Aktuell werden in Kommunikationsprotokollen klassische kryptografische Verfahren genutzt, um die Kommunikation zwischen Geräten und Servern zu verschlüsseln und zu signieren. Es wird davon ausgegangen, dass Quantencomputer in wenigen Jahren so weit entwickelt sind, dass Angreifer diese nutzen, um bei der Kommunikation kritischer Infrastrukturen, wie z.B. Windkraftanlagen, mit ihren Servern, die versendeten Nachrichten zu entschlüsseln. Das BSI arbeitet aktuell unter der Hypothese (in Bezug auf den Hochsicherheitsbereich), dass Quantencomputer Anfang der 2030er-Jahre zur Verfügung stehen [7].

Die Motivation dieser Arbeit liegt darin, dass die Sicherheit von Applikationen und Protokollen im IoT-Bereich auch in Zukunft gewährleistet werden soll. Um die Sicherheit wei-

terhin gewährleisten zu können, müssen die aktuellen Verschlüsselungsverfahren überarbeitet oder durch neue quantensichere Algorithmen ersetzt werden. Das National Institute of Standards and Technology (NIST) hat daher im Jahr 2016 zu einem Standardisierungsverfahren für quantensichere Public-Key-Verschlüsselungsalgorithmen und digitale Signaturen aufgerufen [12]. Nach der dritten Runde wurden bereits ein Public-Key-Verschlüsselungsalgorithmus/Key Encapsulation Mechanism (KEM) und drei Verfahren für digitale Signaturen als Finalisten festgelegt und keine weiteren wurdne in die vierte Runde aufgenommen. Diese vier Algorithmen sollen im Jahr 2023 standardisiert werden. Des Weiteren wurden vier Public-Key-Verschlüsselungsalgorithmen/KEMs in die vierte Runde übernommen. [13] Die Kandidaten in Runde vier werden weiterhin untersucht und auf Schwachstellen überprüft. Es sollen jedoch noch weitere mögliche Standards für digitale Signaturverfahren geprüft werde, sodass ein weiterer *Call for Proposals* [13] durchgeführt wurde.

Das Problem bei den einzuführenden Standards ist, dass diese nicht ohne Anpassungen in bestehende Implementierungen eingefügt werden oder bestehende Code-Bausteine ersetzen können. Es müssen viele verschiedene Anforderungen der unterschiedlichen verwendeten Geräte und Systeme überprüft werden - und das nicht nur im Bereich IoT. Das Konzept „Store now, decrypt later“ [7] zeigt die Dringlichkeit der Forschung zur Post-Quantum-Kryptographie (PQC) auf. Bei diesem Konzept werden aktuelle Kommunikationen verschlüsselt gespeichert, um sie später von Quantencomputern entschlüsseln lassen und rückwirkend mitzulesen zu können [7].

In der heutigen Zeit sind kritische Infrastrukturen laut BKA beliebte Angriffsziele [10]. Die Motivation dieser Arbeit besteht des Weiteren darin, eine quantensichere Alternative für die Wiederherstellung von IoT-Geräten, die auch für kritische Infrastrukturen genutzt werden, nach einer Kompromittierung zu finden.

1.2. Anwendungsfall (Use Case)

Es gibt viele verschiedene Anwendungsfälle bei denen Cyber-Resilienz und speziell die Wiederherstellung eines vertrauenswürdigen und funktionierenden Zustands nach einem Angriff auf ein System relevant sind. Das Projekt *SecDER* [14] des Fraunhofer SIT in Kooperation mit dem Fraunhofer Institut für Energiewirtschaft und Energiesystemtechnik sowie Decoit GmbH, Enertrag AG, Trust at HSH und Ane GmbH & Co. KG beschäftigt sich mit der Erkennung von und Wiederherstellung nach Cyber-Angriffen auf bzw. technische Störungen von dezentralen Energieanlagen und virtuellen Kraftwerken.

Ein weiterer Anwendungsfall könnte das Projekt IMMUNE sein, bei dem das Projektziel darin liegt, „ein Immunsystem für zukünftige industrielle Produktionsinfrastrukturen in der Form von sich selbst verteidigenden Netzwerken umzusetzen“ [15] im Bereich Industrial IoT und Industrie 4.0 [15].

Diese Arbeit soll dazu beitragen dem Projekt SecDER [14] und IMMUNE [15] Möglichkeiten aus dem IoT-Bereich aufzuzeigen. Ein möglicher Anwendungsfall könnte ein intelligentes Energiesystem am Beispiel von Windkraftanlagen sein.

1.3. Stand der Forschung

Es wird in allen Bereichen geforscht, in denen diese Masterarbeit anzusiedeln ist. Zunächst werden einige verwandte Arbeiten zu Cyber-Resilienz im Bereich IoT und ihren Bausteinen wie z.B. Watchdog-Timern vorgestellt. Anschließend wird die Forschung im Bereich der Post-Quanten-Kryptografie und diverse Arbeiten zu quantensicheren Protokollen vorgestellt.

Das NIST hat im Jahr 2018 Richtlinien zu *Platform Firmware Resiliency* veröffentlicht [16]. Darin wurden drei grundlegende Konzepte vorgestellt: Protection, Detection und Recovery. Eine Arbeitsgruppe der Trusted Computing Group (TCG) arbeitet an Anforderungen, die von cyber-resilienten Modulen und Bausteinen [17] erfüllt werden sollen, und spezifizieren Bausteine, wie z.B. Latches oder Watchdog-Timer, die bei der Wiederherstellung eingesetzt werden können.

Die Arbeit von Jin *et al.* [18] setzt sich mit sicheren und cyber-resilienten Microgrids auseinander, die mithilfe von Software-definiertem Networking umgesetzt werden sollen. Hier geht es hauptsächlich um die Erkennung von Angriffen auf Microgrids.

Xu *et al.* [3] stellen in ihrem Paper ein System namens *CIDER* vor, das IoT-Geräte schnell wiederherstellen kann, auch wenn der Angreifer die Root-Kontrolle des Geräts erhalten hat. Der Administrator kann das Gerät nach der Erkennung einer Kompromittierung remote wiederherstellen, indem er ein aktualisiertes Firmware-Image erstellt und CIDER anweist, das Gerät zurückzusetzen, die Firmware zu installieren und damit das Gerät neu zu booten. Xu *et al.* [3] testen *CIDER* mit drei verschiedenen IoT-Plattformen unterschiedlicher Preis- und Leistungsklasse. Sie fanden heraus, dass der Leistungs-Overhead von *CIDER* generell vernachlässigbar ist [3].

Es gibt einige Industriestandards, die gleichbedeutend sind mit dem Begriff *Dominance* [3]. Diese Standards ermöglichen die Fernkonfiguration und das Fernmonitoring von Servern, die allerdings nicht auf den IoT-Bereich angewendet werden können, da sie mehr Hardware- und Softwareanforderungen erfüllen müssen als es bei IoT-Geräten möglich ist. [3]

Huber *et al.* [2] entwarfen das cyber-resiliente System *Lazarus*, das die Fernwiederherstellung von kompromittierten IoT-Geräten in einer festgelegten Zeitspanne ermöglicht. Dies geschieht mithilfe eines Watchdog-Timers, der in einer TEE ausgeführt wird. Des Weiteren wird eine zeitliche und räumliche Trennung der Trusted Computing Base (TCB) von der nicht-vertrauenswürdigen Software durch die Verwendung der TEE sichergestellt [2]. Durch diese zeitliche Isolierung sind Angriffe auf Seitenkanäle zwecklos, da sie keine Daten leaken können. Außerdem verwenden Huber *et al.* [2] eine erweiterte Form der Device Identifier Composition Engine (DICE), um festzustellen, ob Updates installiert wurden, in dem die Geräteidentität auch nach einem Update und mit einem neuen öffentlichen Schlüssel noch

erfolgreich zugeordnet werden kann [2]. Lazarus ist hauptsächlich auf einfache, kostengünstige IoT-Geräte ausgelegt, was auch auf einem ARM-Cortex-M33-basierten Mikrocontroller getestet wurde [2]. Wie auch bei *CIDER* [3] sind die Auswirkungen auf die Laufzeitleistung vernachlässigbar [2].

Die Arbeit von Medwed *et al.* [19] beschäftigt sich mit der Cyber-Resilienz von IoT-Geräten, die sich selbst überwachen. Dabei soll eine große Anzahl an IoT-Geräten aus der Ferne rechtzeitig wiederhergestellt werden können, ohne dass ein langer Ausfall sich negativ auf die Geräte auswirkt oder diese physisch zerstört. Bei diesem erweiterten Ansatz gehen Medwed *et al.* [19] davon aus, dass die IoT-Geräte auch physisch zerstört werden könnten. Speziell wird dabei auf verschiedene Bausteine zur Angriffserkennung und Wiederherstellung eingegangen.

In der Veröffentlichung von Jäger *et al.* [20] geht es darum die Resilienz mithilfe eines eingebetteten Netzwerk-Knotens in den Bereich Industrial IoT einzuführen. Trusted Platform Modules (TPMs) und Remote Attestation werden verwendet, um Angriffe zu detektieren und davon zu berichten. Ein Authenticated Watchdog Timer (AWDT) hilft dabei die Plattform in einen unkompromittierten Zustand zurückzubringen. Die Kombination soll dazu dienen, dass die Ausfallsicherheit auf Plattformebene gewährleistet ist. Sie soll auch als Grundlage zur Ausfallsicherheit von Netzwerken dienen. Abschließend wird der Ressourcenverbrauch und die Performance des Netzwerk-Knotens auf Eignung im industriellen IoT-Umfeld überprüft. [20]

Des Weiteren wird sich damit beschäftigt wie bestehende Systeme und Protokolle quantensicher gemacht werden können.

Bisher gibt es einige Arbeiten, die die Möglichkeiten betrachten Post-Quantum-Kryptographie (PQC) in verschiedene Protokolle wie OpenVPN und Transport Layer Security (TLS) in OpenSSL [21] zu integrieren. Dazu gehört u.a. die Arbeit von Bürstinghaus-Steinbach *et al.* [22], die die Algorithmen Kyber und SPHINCS+ in die eingebettete TLS Bibliothek *mbed TLS* integriert. Anschließend wurde die Performance der quantensicheren Primitive auf vier unterschiedlichen eingebetteten Plattformen mit verschiedenen Prozessoren gemessen. Außerdem wurde die Performance der quantensicheren TLS-Varianten mit den klassischen TLS-Varianten, die auf elliptischen Kurven aufbauen, miteinander verglichen. Das Ergebnis zeigte, dass die quantensichere Schlüsseleinrichtung mit Kyber performant lief im Vergleich zu den klassischen Varianten der Elliptische Kurven Kryptografie. Bei der Verwendung des Signaturverfahrens SPHINCS+ wurde festgestellt, dass eingebettete Systeme besser als PQC-TLS-Clients geeignet sind als als Server, da die Signaturgrößen und die Zeit für die Erstellung der Signatur hoch für eingebettete Systeme sind. [22]

Auch Hülsing *et al.* forschten an der Verwendbarkeit von SPHINCS-256 für eingebette Systeme im Vergleich zu Extended Merkle Signature Scheme (XMSS). Dabei wurde herausgefunden, dass obwohl die SPHINCS-Signatur nicht in den Speicher des IoT-Geräts passt, die Ausführung von SPHINCS-Signaturen im Hinblick auf die Performance und den Speicherverbrauch auf IoT-Geräten machbar ist. Schlussfolgerung von Hülsing *et al.* ist, dass SPHINCS-256 eine gute Möglichkeit für ein quantensicheres Cross-Plattform-Signaturverfahren ist. [23]

Stebila und Mosca [24] betrachten zwei Protokolle, die jeweils einen quantenresistenten Schlüsselaustausch auf gitterbasierten Problemen beinhalten. Sie erörtern u.a. deren Sicherheit sowohl im Kontext von TLS als auch einzeln. Des Weiteren führen sie das Projekt Open Quantum Safe [25] ein. Dabei handelt es sich um ein Open-Source-Software-Projekt, das die Implementierung von Prototypen mit quantenresistenter Kryptografie ermöglicht. Dies inkludiert außerdem eine C-Bibliothek [26], die verschiedene quantensichere Algorithmen zur Verfügung stellt.

Duits [27] hat in ihrer Masterarbeit das Signal-Protokoll für Chat-Applikationen untersucht, das aktuell die ellitptische Kurve *Curve25519* für den Schlüsseltausch und die Schlüsselableitungsfunktion SHA-256 als kryptografische Primitiven verwendet. Die elliptische Kurve wird durch 44 verschiedene Versionen von zehn verschiedenen quantensicheren KEMs und dem Algorithmus Supersingular Isogeny Diffie–Hellman Key Exchange (SIDH) ersetzt. Duits [27] hat einen Großteil der Algorithmen der Open-Quantum-Safe-Bibliothek [26] implementiert, um zu evaluieren, wie der jeweilige Algorithmus die Performance des Signal-Protokolls in Bezug auf Laufzeit (CPU-Zyklen), Bandbreite, Energieeffizienz sowie Speicherplatzanforderungen beeinflusst. Des Weiteren hat Duits [27] verschiedene teilweise quantensichere Signal-Protokolle, die einfacher zu implementieren sind, analysiert.

Im Workshop *Identifying Research Challenges in Post Quantum Cryptography Migration and Cryptographic Agility* [28] wurde sich damit auseinandergesetzt, wie die Migration von klassischer zu quantensicherer Kryptografie erfolgen und wie agile Kryptografie entwickelt werden kann. Es wurde herausgefunden, dass die Ersetzung der implementierten Algorithmen durch quantensichere Algorithmen in allen möglichen Systemen einen großen Forschungsbedarf hat, um zu verstehen und zu quantifizieren wie sich diese Ersetzungen auf das jeweilige System auswirken können [28]. Da PQC-Algorithmen z.B. mehr Speicherplatz und eine höhere Rechenleistung erfordern, muss entsprechend geforscht und prototypisiert werden [28]. Des Weiteren sollte die Einführung von hybriden Systemen erforscht werden, um quantensichere Kryptografie in laufende Systeme zu integrieren, die nicht gestoppt werden sollten, um neue Software aufspielen zu können [28].

Das Projekt SAFEcrypto ist ein Akronym für *Secure Architectures of Future Emerging Cryptography* [29]. Bei SAFEcrypto geht es um die Bereitstellung einer neuen „Generation praktischer, robuster und physisch sicherer Post-Quantum-Kryptographielösungen“ [29], die die Sicherheit von Systemen der Informations- und Kommunikationstechnik sowie deren Dienste und Anwendungen langfristig gewährleisten sollen. Hauptsächlich werden Public-Key-Verfahren basierend auf dem Gitterproblem entwickelt und analysiert. Des Weiteren werden Analysen durchgeführt und Schutzmethoden entwickelt, die vor physischen Angriffen auf Hard- und Softwareimplementierungen, die gitterbasierte Kryptografie nutzen, schützen sollen. Außerdem wird die Schlüsselverteilung betrachtet. [29]

Open Quantum Safe (OQS) ist ein weiteres Open-Source-Projekt, bei dem eine Bibliothek mit einem Großteil der beim NIST eingereichten PQC-Algorithmen zusammengetragen wurde [30]. Mit dieser Bibliothek wird ein Application Programming Interface (API) zum Testen und Prototyping dieser PQC-Algorithmen bereitgestellt. Die Bibliothek ist in C implementiert - da die Referenzimplementierungen für den NIST-Wettbewerb in C eingereicht werden mussten. OQS stellt jedoch Bindings für die Programmiersprachen Rust, Go,

C++, Python, .NET und Java in ihrem GitHub-Repertoire [31] zur Verfügung. Auch Duits [27] hat in ihrer Thesis mit der Bibliothek `liboqs` gearbeitet. Weitere bekannte Projekte und Bibliotheken, die mit `liboqs` arbeiten, sind OpenSSL [32] und OpenSSH [33].

PQCrypto ist ein Open-Source-Projekt, für das Wissenschaftler weltweit arbeiten und Ideen und Algorithmen in Bezug auf Post-Quanten-Kryptografie entwickeln. Aus dem PQCrypto-Projekt ist die kryptografische Software-Bibliothek `libpqcrypto` [34] entstanden. Die Bibliothek besitzt jeweils eine API für Python und C, kann aber auch von der Kommandozeile aus ausgeführt werden. Die Bibliothek enthält 19 der 22 beim NIST-Standardisierungsprozess eingereichten Algorithmen, insgesamt jedoch 77 kryptografische Systeme.

Es gibt noch viel mehr Forschung in den genannten Bereichen. Weitere Nennungen würden jedoch den Rahmen dieser Arbeit überschreiten.

1.4. Ziele der Arbeit

Im Folgenden werden die Forschungsziele der Masterarbeit genauer definiert (Abschnitt 1.4.1) sowie die Forschungsfragen aufgestellt (Abschnitt 1.4.2). In Abschnitt 1.4.3 wird das Thema der Arbeit eingegrenzt.

1.4.1. Forschungsziele

Ziel dieser Arbeit ist es, die im *Lazarus*-System [2] verwendeten Bausteine zu betrachten und analysieren, welche Bausteine ein neu konzipiertes Watchdog-Timer-Protokoll verwenden soll, um quantensicher zu sein. Mithilfe dieses quantensicheren Protokolls soll die Cyber-Resilienz durch die Wiederherstellung eines sicheren, vertrauenswürdigen Zustands eines IoT-Geräts erhöht werden.

In der Masterarbeit werden folgende Ziele und die damit verbundene Konzeptionierung, Implementierung und Evaluierung betrachtet:

- Es soll aufbauend auf der Forschung von Huber *et al.* [2] ein auf dem Lazarus-System und deren Prototyp [35] basierendes Watchdog-Timer-Protokoll zur Wiederherstellung eines IoT-Geräts konzeptioniert und als Proof-of-Concept-Implementierung in Software umgesetzt werden.
- Es soll überprüft werden, ob das neu konzipierte Watchdog-Timer-Protokoll unter Verwendung quantensicherer Algorithmen theoretisch und praktisch als Proof-of-Concept-Implementierung in Software umsetzbar ist. Dazu werden folgende Schritte durchgeführt:
  - Eine Analyse der in den CIDER- [3] und Lazarus-Protokollen [2] verwendeten Bausteine.

- Eine Analyse der klassischen Bausteine des neu konzipierten Watchdog-Timer-Protokolls, die gegen quantensichere Bausteine ausgetauscht werden müssen, um das Protokoll quantensicher zu gestalten.

- Eine Übersicht und Analyse verschiedener Eigenschaften der zu standardisierenden PQC-Algorithmen.

- Eine Proof-of-Concept-Implementierung des klassischen und quantensicheren Watchdog-Timer-Protokolls.

- Die Ermöglichung der Nutzung verschiedener klassischer und quantensicherer Algorithmen in der Implementierung des Watchdog-Timer-Protokolls.

- Eine Durchführung und Auswertung von Messungen bezüglich des Ressourcenbedarfs von Python-Wrapper-Funktionen verschiedener PQC-Algorithmen als Vergleich zu Funktionen, die klassische Algorithmen nutzen.

- Eine Durchführung und Auswertung von Messungen des Ressourcenbedarfs unterschiedlicher Abläufe des neu konzipierten Watchdog-Timer-Protokolls.

- Eine Einschätzung über die Spezifikationen, die IoT-Geräte benötigen, um das neu konzipierte Watchdog-Timer-Protokoll verwenden zu können.

1.4.2. Forschungsfragen

Die folgenden Forschungsfragen werden in der Masterarbeit behandelt und in Abschnitt 7.3 final beantwortet:

1. Wie kann ein Watchdog-Timer-Protokoll unter Verwendung klassischer Kryptografie umgesetzt werden, um ein IoT-Gerät wiederherzustellen?

2. Wie kann ein Watchdog-Timer-Protokoll unter Verwendung quantensicherer Kryptografie umgesetzt werden, um ein IoT-Gerät wiederherzustellen?

3. Wie hoch ist die Performance-Unterschied (in CPU-Zyklen) zwischen den Funktionen ausgewählter PQC-Algorithmen im Vergleich zu den Funktionen ausgewählter klassischer Algorithmen unter der Verwendung von Python-Wrappern?

4. Wie hoch ist der Performance-Unterschied (in CPU-Zyklen) zwischen einem quantensicheren Watchdog-Timer-Protokoll im Vergleich zu einem klassischen Watchdog-Timer-Protokoll für unterschiedliche Szenarien?

5. Welche Eigenschaften sollte ein IoT-Gerät besitzen, um das quantensichere Watchdog-Timer-Protokoll nutzen zu können?

1.4.3. Abgrenzung

In dieser Arbeit liegt der Fokus auf der quantensicheren Kommunikation zwischen einem Server und einem IoT-Gerät. Für die Kommunikation soll ein neu konzipiertes Watchdog-Timer-Protokoll, das mit digitalen Signaturen arbeitet, verwendet werden. In dieser Arbeit wird das IoT-Gerät in der Konzeption (Kapitel 3) als *Gerät* und in der Implementierung (Kapitel 4) als `Device` bezeichnet.

Der Lebenszyklus eines solchen Systems kann aus drei Phasen bestehen:

1. Device Provisioning (Gerätebereitstellung) - Verteilung der IoT-Geräte mit Anschluss an den Server sowie Verteilung der öffentlichen Schlüssel von Server und Gerät
2. System im Einsatz mit verschiedenen möglichen Szenarien
3. Trennung eines Geräts von einem Server (Entfernung aus einem System)

Die Phase der Gerätebereitstellung und der Entfernung von IoT-Geräten aus einem System werden nicht betrachtet. Dazu zählt u.a. die Änderung des Geräteeigentümers, wofür z.B. ein neuer öffentlicher Server-Schlüssel bereitgestellt sowie der öffentliche Schlüssel des Geräts an den neuen Server ausgehändigt werden müsste.

Die drei Prinzipien, die zur Cyber-Resilienz gehören sind: Erkennung, Schutz und Wiederherstellung [36](vgl. Abschnitt 2.1.2). In dieser Arbeit wird ausschließlich die Wiederherstellung von IoT-Geräten betrachtet. Des Weiteren wird davon ausgegangen, dass erkannt wird, wann und ob ein Gerät kompromittiert oder eine Schwachstelle gefunden wurde. Neben der (quantensicheren) Verschlüsselung werden keine weiteren Schutzmaßnahmen berücksichtigt.

Da bei der Implementierung des Lazarus-Systems in Hardware einige Probleme auftreten könnten und der Fokus in dieser Arbeit nicht auf der Hardware liegt, wird auf eine Hardware-Implementierung verzichtet. Um ein besseres Verständnis zum Ablauf des Protokolls zu bekommen und möglichst kostengünstig eine funktionierende Implementierung zu realisieren, wird eine Proof-of-Concept-Implementierung basierend auf dem Prototyp von Huber *et al.* [2] [35] ausschließlich in Software nachgebaut. Hierbei wird außerdem darauf verzichtet, dass die kritischen Abschnitte (wie z.B. der Watchdog-Timer) innerhalb einer TEE laufen sollen, da dies vorausgesetzt werden kann.

Ein Fokus dieser Arbeit liegt auf der Resilienz, weshalb der Watchdog-Timer einen Hauptteil einen großen Teil der Konzeption und Implementierung einnimmt. Remote Attestation als Methode der Angriffserkennung wird nicht betrachtet.

In dieser Arbeit wird davon ausgegangen, dass nur die asymmetrischen Verfahren gebrochen werden können. Es werden daher ausschließlich quantensichere, digitale Signaturverfahren berücksichtigt. Die Erstellung und Verteilung von Schlüsseln soll provisorisch erfolgen, sodass das Signieren und Verifizieren von Nachrichten bzw. Signaturen umgesetzt werden kann. Um die Szenarien zu implementieren und die digitalen Signaturverfahren nutzen zu können, werden im Vorfeld Schlüssel erstellt und diese mithilfe des Dateisystems an die

richtigen Speicherorte gelegt (vgl. Abschnitt 4). Es wird davon ausgegangen, dass die Zertifikate mit den öffentlichen Schlüsseln sicher bzw. quantensicher sind - je nachdem welches Signaturverfahren zur Schlüsselerstellung verwendet wurde. Des Weiteren wird angenommen, dass der Server sicher und verfügbar ist.

Alle aufgeführten Exklusionen gehen über den Rahmen dieser Arbeit hinaus, können aber als Grundlage für weiterführende Forschung (siehe Diskussionskapitel 6 und Abschnitt 7.4) dienen.

1.5. Methodik und Vorgehensweise

In diesem Kapitel werden die in dieser Arbeit verwendeten Methoden mit ihren jeweiligen Vorgehensweisen vorgestellt. Die Technology Readiness Levels (TRLs) (Abschnitt 1.5.1) sowie eine Kombination induktiver und quantitativer, sowie qualitativer Forschungsmethoden mithilfe der Durchführung von Messreihen (Abschnitt 1.5.2) wird gewählt und erläutert.

1.5.1. Technology Readiness Level

Die Entwicklung eines generischen, quantensicheren Watchdog-Timer-Protokolls kann als Entwicklung einer neuen Technologie bezeichnet werden. Um das Protokoll zu entwickeln, werden die Stufen 1 bis 3 der Technology Readiness Level (TRL) nacheinander abgearbeitet. Bei den TRLs handelt es sich um eine Methode, mit der die technische Reife einer Technologie angegeben werden kann [37]. So kann festgestellt werden wie weit fortgeschritten die Forschung und Entwicklung einer Technologie ist. Ursprünglich nutzte die National Aeronautics and Space Administration (NASA) die TRL, die Europäische Union (EU) hat die Definitionen jedoch normalisiert. Die TRL setzen sich aus neun Stufen [37] zusammen, von denen in dieser Arbeit die ersten drei Stufen betrachtet werden.

1. Grundlegende Prinzipien beobachtet und beschrieben
2. Technologie-Grundprinzipien beobachtet und beschrieben - Konzept formuliert
3. Konzept experimentell bewiesen

1.5.1.1. TRL 1: Grundlegende Prinzipien beobachten

In dieser Arbeit wird Stufe 1 mithilfe der Methode *Literaturrecherche* durchgeführt. Aus den Beobachtungen, die in der Einleitung (Kapitel 1) dargestellt wurden, hat sich das Thema der Masterarbeit herauskristallisiert. Der aktuelle Forschungsstand (Abschnitt 1.3) und die relevanten Technologien, sowie deren Grundlagen (Abschnitt 2) werden herausgearbeitet. Zuvor werden die Ziele der Masterarbeit und die Forschungsfragen (Abschnitt 1.4) beschrieben. Das Thema der Masterarbeit wird von bereits bestehenden Konzepten abgegrenzt und eingeordnet (Abschnitt 1.4.3).

1.5.1.2. TRL 2: Technologiekonzept formulieren

Die zweite Stufe erfolgt basierend auf den Ergebnissen der Literaturrecherche aus Stufe 1. In Bezug auf TRL 2 wird ein Konzept formuliert, das die Technologie eines Watchdog-Timer-Protokolls beschreibt.

Das Konzept basiert auf der Grundlage zweier Paper [3] [2] und einem Lazarus-Prototyp [35]. Die genannten Paper verwenden standardisierte klassische Signaturalgorithmen. Im Zuge der Konzeption (Kapitel 3) werden die in den Papern beschriebenen Bausteine, Algorithmen und Nachrichtentypen eines klassischen Watchdog-Timer-Protkolls (Abschnitt 3.1) herausgearbeitet.

Das Protokoll wird evaluiert und auf die Funktionen und Aufgaben, die das neu zu konzipierende Protokoll erfüllen soll, angepasst. Das neu entwickelte Konzept ist eine vereinfachte Version der bereits bestehenden Protokolle CIDER [3] und Lazarus [2]. Die Bausteine der Protokolle werden zum Teil in das neue Konzept übertragen. Daraufhin wird der Gesamtablauf des klassischen Protokolls (Abschnitt 3.1.2) herausgearbeitet. Der Ablauf soll möglichst gleich bleiben, kann aber etwas abgewandelt werden, da durch die bestehende Forschung schon einige Aspekte als funktionstüchtig angesehen werden können (Abschnitt 3.2.1). Die unterschiedlichen Abläufe des Protokolls aufgrund verschiedener Zustände werden als Szenarien in Abschnitt 3.1.4 mithilfe von Sequenzdiagrammen dargestellt. Die Sequenzdiagramme wurden mit der Online-Software diagrams.net (ehemals: draw.io) erstellt.

Um einen neuen Ansatz für ein quantensicheres Watchdog-Timer-Protokoll zu erreichen, wird das bereits bestehende Konzept für das klassische Watchdog-Timer-Protokoll betrachtet. Es wird analysiert, welche Bausteine und Algorithmen ausgetauscht werden müssen, um das Protokoll quantensicher zu machen (Abschnitt 3.2.2). Der im klassischen Watchdog-Timer-Protokoll verwendete Algorithmus *secp256r1* wird aus dem Lazarus-Prototyp [35] übernommen und es werden zusätzlich noch die klassischen Verfahren RSA2048 und RSA4096 eingesetzt und getestet. Die Formulierung des Konzepts beinhaltet außerdem eine Festlegung der zu testenden PQC-Algorithmen.

1.5.1.3. TRL 3: Experimenteller Nachweis des Konzepts

Bei den Forschungsfragen 1 und 2 (Abschnitt 1.4.2) soll herausgefunden werden, ob die Umsetzung eines klassischen und quantensicheren Watchdog-Timer-Protokolls möglich wäre.

Der experimentelle Nachweis des Konzepts erfolgt in dieser Arbeit anhand eines Proof-of-Concepts in Software des in Stufe 2 formulierten Konzepts. Die Implementierung des Proof-of-Concepts erfolgt in der Programmiersprache Python und wird mithilfe des Aktor-Frameworks `Thespian` umgesetzt. Dazu werden mithilfe einer Literaturrecherche, mögliche zu verwendende Python-Bibliotheken bestimmt (Abschnitt 4.2).

Die Implementierung des Konzepts erfolgt schrittweise. Als Erstes wird das bestehende Konzept als Grundlage für ein zunächst unverschlüsseltes Watchdog-Timer-Protokoll verwendet. Dies dient der Übersicht über die generellen Abläufe und den Versand von Nachrichten zwischen Gerät und Server. Es wurde mit einzelnen Aktoren begonnen, die sich Nachrichten senden und nach und nach werden weitere Aktoren implementiert.

Die Funktionen und Aufgaben der einzelnen Aktoren werden nacheinander ergänzt. Es wird darauf geachtet, dass die Funktionen für die Erstellung und Verifizierung der digitalen Signatur austauschbar sind, sodass generisch noch weitere Bibliotheken verwendet werden können. Der genaue Ablauf und Prozess der Implementierung wird in den Abschnitten 4.4 und 4.5 beschrieben.

Die Proof-of-Concept-Implementierung des vom Lazarus-Protokoll abgewandelten Watchdog-Timer-Protokolls dient zur teilweisen Validierung des Lazarus-Protokolls [2] in der Skriptsprache Python. Somit kann die klassische und die quantensichere Version des Protokolls miteinander verglichen werden. Mithilfe des Proof-of-Concepts soll überprüft werden, ob es möglich ist quantensichere Algorithmen in das bestehende Protokoll funktionsgemäß zu implementieren. Die Proof-of-Concept-Implementierung wird jeweils mit unterschiedlichen Systemzuständen getestet. So wird überprüft, ob alle möglichen Schritte bzw. Szenarien, die ein solches System durchlaufen kann, erfolgreich implementiert wurden.

Durch die Umsetzung der dritten TRL-Stufe soll qualitativ nachgewiesen werden, dass die Abläufe und Szenarien des Watchdog-Timer-Protokolls sowohl mit klassischen als auch mit quantensicheren Algorithmen als kryptografische Bausteine umgesetzt werden können.

1.5.2. Forschungsmethode

Um die Forschungsfragen 3, 4 und 5 (vgl. Abschnitt 1.4.2) beantworten zu können, wird eine induktive Forschungsmethode in Kombination mit einer quantitativen und qualitativen Forschungsmethode gewählt. Bei einer induktiven Forschung geht es um die Erstellung von Theorien [38]. Die neue Theorie bezieht sich auf die Beantwortung der genannten Forschungsfragen.

Anhand der durchzuführenden Messreihen soll die Anzahl der jeweils benötigten CPU-Zyklen unter der Verwendung der klassischen und quantensicheren Algorithmen der Python-Bibliotheken bestimmt werden, da es dafür keine Benchmarks gibt. Diese Messreihen dienen der Beantwortung der Forschungsfrage 3. Durch die Messung der CPU-Zyklen der Funktionen kann im besten Fall auf die CPU-Zyklen der Szenarien, bei denen die Signier- und Verifizierfunktionen Einsatz finden, geschlossen werden.

Die induktive Forschung baut auf dem in Schritt 3 (vgl. Abschnitt 1.5.1.3) implementierten Protokoll auf. Es wird induktiv geforscht, welche Auswirkungen die Ersetzung von klassischen kryptografischen Primitiven durch quantensichere Primitive auf das Protokoll und seine Eigenschaften hat. Es soll mithilfe der durchzuführenden Messreihen unter

Verwendung verschiedener PQC-Algorithmen herausgefunden werden, welcher Algorithmus sich für den Einsatz des in Kapitel 3 konzipierten und in Kapitel 4 implementierten Watchdog-Timer-Protokolls eignet. Dafür werden die CPU-Zyklen ausgewählter Algorithmen beim Durchlaufen verschiedener Szenarien des Protokollablaufs gemessen. Die für die Messungen gewählten quantensicheren, digitalen Signaturalgorithmen sind die Gewinner des PQC-Standardisierungsprojekts des NIST. Im Vergleich dazu werden die klassischen Algorithmen RSA2048, RSA4096 und secp256r1 gewählt. Anhand der Messergebnisse der Szenario-Messreihen soll eine Generalisierung für das Watchdog-Timer-Protokoll abgeleitet werden.

Die Durchführung der Szenario-Messreihen kann auch als sogenannte Mehrfachfallstudie angesehen werden und zählt somit als qualitative Forschungsmethode [39]. Bei Mehrfachfallstudien kann laut Saric [40] ein Vergleich über die Ergebnisse mehrerer Fälle gezogen werden. Dabei wird das Vertrauen in die finalen Ergebnisse erhöht und eine Grundlage für die Bildung von Theorien geschaffen. Auch im Falle von Replikation kann bestimmt werden, ob sich theoretische Aussagen bestätigen. Durch den Vergleich können entstehende Theorien verallgemeinert werden. Bei der Planung solcher Fallstudien muss festgelegt werden, was genau der eigentliche Fall ist. Der Fall sollte von den zu beantwortenden Forschungsfragen abhängen. [40] Bei einer Fallstudie wird empirisch vorgegangen und es wird ein Fall aus mehreren Blickwinkeln betrachtet [39]. In Bezug auf das Watchdog-Timer-Protokoll werden die Fälle durch die verschiedenen Abläufe des Protokolls als Szenarien hergeleitet (vgl. Abschnitt 3.1.4).

Die quantitative Forschungsmethode beruht auf der statistischen Auswertung der durchgeführten Messreihen (vgl. Abschnitte 5.4.2 und 5.5.2). Es sollen Mediane, Mittelwerte, Standardabweichung, niedrigste und höchste Werte der Messreihen unter Zuhilfenahme des Python-Packages `numpy` bestimmt werden. Der Vergleich und die Bestimmung der Unterschiede zwischen den Messergebnissen für die verschiedenen Algorithmen wird mithilfe von Excel durchgeführt.

Auf Grundlage der ausgewerteten Messergebnisse sowie den Schlüssel- und Signaturgrößen, die bei den verschiedenen Algorithmen erstellt werden, soll die Forschungsfrage 5 (Abschnitt 1.4.2) mit dem Aufstellen einer neuen Theorie beantwortet werden.

Es wird darauf geachtet, dass die Gütekriterien der qualitativen und quantitativen Forschung erfüllt werden. Zu den Gütekriterien qualitativer Forschung zählen: Transparenz, Reichweite und Intersubjektivität [41] . Validität, Reliabilität, Variierbarkeit, Objektivität und Planbarkeit gehören zu den Gütekriterien für gültige Experimente [42]. Die Messungen der Funktionen und Szenarien werden in dieser Arbeit anhand dieser Gütekriterien bewertet. Inwiefern die Gütekriterien während der Durchführung der Masterarbeit eingehalten wurden, wird in den Abschnitten 6.4.1 und 6.4.2 diskutiert.

1.6. Struktur der Arbeit

Im weiteren Verlauf der Masterarbeit werden zuerst die Grundlagen zu Informationssicherheit, der klassischen und quantensicheren Kryptografie sowie dem Thespian-Framework in Kapitel 2 zusammengefasst. In Kapitel 3 wird das abgewandelte Watchdog-Timer-Protokoll konzipiert und der Ablauf verschiedener möglicher Szenarien und wichtige Nachrichtentypen beschrieben. Die Proof-of-Concept-Implementierung wird in Kapitel 4 vorgestellt. Anschließend erfolgen die Messungen und die Auswertung der Messreihen in Kapitel 5. In Kapitel 6 werden die Ergebnisse der Implementierung und der Messungen sowie benötigte Eigenschaften zur Anwendung des Watchdog-Timer-Protokolls im IoT-Bereich diskutiert. Außerdem werden in diesem Kapitel die Forschungsfragen final beantwortet sowie ein Ausblick auf mögliche weitergehende Forschung gegeben. In Kapitel 7 werden die Ergebnisse der Arbeit abschließend zusammengefasst.

2. Grundlagen

Dieses Kapitel liefert zuerst eine Erklärung des Begriffs Informationssicherheit (Abschnitt 2.1) und die dazugehörigen Schutzziele (Abschnitt 2.1.1). Anschließend folgt eine Erläuterung zum Thema Cyber-Resilienz (Abschnitt 2.1.2). In Abschnitt 2.2 wird das Aktor-Framework vorgestellt und in Abschnitt 2.3 eine Einführung in die Kryptografie gegeben, gefolgt von Erläuterungen zur klassischen Kryptografie (Abschnitt 2.3.1) sowie der Post-Quantum-Kryptographie (PQC) (Abschnitt 2.4).

2.1. Informationssicherheit

Das National Institute of Standards and Technology (NIST) und das Bundesamt für Sicherheit in der Informationstechnik (BSI) setzen sich mit dem Thema IT-Sicherheit auseinander. Zu den Aufgaben des BSI gehören u.a. „Förderung und Verbesserung der Sicherheit in der Informationstechnik in allen Bereichen“, „Bearbeiten von Grundlagen der IT-Sicherheit“, „Entwicklung von ‚sichere‘ Systemen und Komponenten in Kooperation mit Herstellern und Anwendern“, sowie die „Überprüfung (Evaluierung) und Zertifizierung von Produkten und Systemen nach internationalen Sicherheitskriterien“. [43] Das NIST ist eine Organisation, die Standards publiziert und Empfehlungen für die Verwendung bestimmter Verfahren an die amerikanische Regierung richtet. [44] Das NIST versteht unter dem Begriff Informationssicherheit bzw. IT-Sicherheit den Schutz von Informationen und Informationssystemen vor unautorisiertem Zugriff, sowie der Verwendung, Offenlegung, Unterbrechung, Modifizierung oder Zerstörung dieser, um Vertraulichkeit, Integrität und Verfügbarkeit zu gewährleisten [45].

2.1.1. Schutzziele

In dieser Arbeit werden ausschließlich die Schutzziele Vertraulichkeit, Integrität von Daten und Kommunikationspartnern, sowie die Verfügbarkeit und Authentizität von Daten und Kommunikationspartnern behandelt. Weitere Schutzziele, die in dieser Arbeit nicht betrachtet werden, sind z.B. Nachweisbarkeit/Nicht-Abstreitbarkeit, Privatheit, Vertrauenswürdigkeit und Widerstandsfähigkeit [46].

In Tabelle 2.1 werden technische Maßnahmen und dazugehörige Beispiele aufgeführt, die dafür sorgen können, dass in einem System Schutzziele eingehalten werden können.

Neben den in Tabelle 2.1 aufgeführten technischen Maßnahmen, können auch organisatorische Maßnahmen zum Einhalten der Schutzziele umgesetzt werden. Eine organisatorische

Schutzziel	Technische Maßnahme	Beispiele
Vertraulichkeit	Verschlüsselung	AES, RSA, ChaCha
Integrität	Kryptografische Prüfsummen (Hashes)	SHA-256, SHA-3
Verfügbarkeit	Schutz auf Netzwerkebene, Systemebene	Routing, Firewalls
Authentizität (Daten)	Message Authentication Code (MAC)	CMAC, HMAC
Authentizität (Identität)	Authentifizierung anhand von Wissen, Besitz oder Biometrie	Schlüssel, Passwort, PIN, TAN, Digitale Signatur, Iris, Fingerabdruck

Tabelle 2.1.: Technische Maßnahmen zur Umsetzung von Schutzzielen (mit Beispielen) [46]

Maßnahme zum Einhalten der Integrität ist die Verwaltung und Veröffentlichung von Prüfsummen von Informationen, die mithilfe von Hashwerten von Dokumenten bereitgestellt werden können. Des Weiteren hilft das Management von Identitäten und die 2-Faktor-Authentisierung durch die Verwendung digitaler Signaturen bei der Gewährleistung der Authentizität.

Vertraulichkeit. Bei der Vertraulichkeit soll gewährleistet werden, dass Informationen, die zwischen zwei Parteien ausgetauscht werden, nicht an Dritte weitergegeben werden. Ein Synonym für diese Eigenschaft ist der Begriff Geheimhaltung. Um Vertraulichkeit zu erreichen, können Nachrichten bzw. Informationen verschlüsselt werden. Die Parteien, die die Information im Klartext lesen dürfen, haben einen Schlüssel um die Nachricht zu ver- oder entschlüsseln. Eine dritte Person, die die Nachricht nicht lesen soll, sieht nur eine für sie unlesbare Nachricht, den sogenannten Ciphertext. Die dritte Person darf nicht in der Lage sein an den Schlüssel zu gelangen. [44]

Integrität. Laut Barker [44] soll bei der Integrität gewährleistet werden, dass Daten beim Transport nicht unbemerkt verändert werden. Es sollen weder Informationen hinzugefügt noch ersetzt oder gelöscht werden. Um diese Eigenschaft zu erreichen, werden kryptografische Hashfunktionen (Abschnitt 2.3.2) oder digitale Signaturen (Abschnitt 2.3.1.3) verwendet. Mit ihnen kann geprüft werden, ob Nachrichten verändert wurden, egal ob durch einen Angreifer oder durch Rauschen. Sie dienen als Integritätsschutz.

Verfügbarkeit. Laut NIST [45] handelt es sich bei der Verfügbarkeit um eine Eigenschaft bei der gewährleistet sein soll, dass das Zugreifen und die Verwendung von Informationen bei Bedarf immer möglich ist. Dazu können Backups, Redundanz oder Watchdog-Timer helfen. Durch Watchdog-Timer wird sichergestellt, dass das System schnell wieder verfügbar ist.

Authentizität. Das NIST [45] beschreibt Authentizität als die Eigenschaft, echt zu sein und verifiziert werden zu können sowie vertrauenswürdig zu sein. Authentisierung beschreibt den Nachweis einer Person, dass diese diejenige Person ist, die sie vorgibt zu sein. Um die Authentizität feststellen zu können, werden Authentifizierungsverfahren angewendet, bei denen es sich um einen Prozess zur Überprüfung von Identitäten oder des Ursprungs einer Nachricht sowie die Integrität von Daten handelt. Ein Beispiel wäre das Nachweisen des Besitzes eines Passworts [47]. Die Authentifizierung der Identität berechtigt meist zum Zugriff auf bestimmte Daten. Bei der Nachrichtenauthentizität kann eine empfangende Partei zweifelsfrei nachweisen von welchem Sender die Nachricht stammt [48], z.B. anhand einer Unterschrift, die auch digital sein kann. Darunter zählen z.B. auch Signaturverfahren, für die keine Verifikation von einem Geheimnis notwendig ist [47].

2.1.2. Cyber-Resilienz

Es gibt unterschiedliche Definitionen für den Begriff Resilienz. Bei der Cyber-Resilienz handelt es sich laut TCG um die drei Prinzipien Erkennung, Schutz und Wiederherstellung [36]. In dieser Arbeit wird sich ausschließlich mit dem Thema *Wiederherstellung* beschäftigt.

Die Wiederherstellung eines Systems kann dann relevant sein, wenn ein System ausfällt z.B. durch einen Stromausfall oder einen Hack bzw. eine Manipulation des Systems. Um in einem System Resilienz zu erreichen, werden von der TCG [17] Latches und Watchdog-Timer als Bausteine empfohlen. In dieser Arbeit wird sich ausschließlich mit dem Watchdog-Timer beschäftigt. Weiterführende Literatur zum Thema Latches stellt die TCG unter [17] zur Verfügung.

Beim *Watchdog-Timer* handelt es sich um einen Mechanismus, der ein Signal auslöst oder eine Aktion durchführt, wenn ein Zähler abgelaufen ist. Der Ablauf des Zählers kann hinauszögert werden, in dem er wieder um eine gewisse Zeitspanne erhöht wird, dies wird *bedienen*/to service genannt. Wenn der Watchdog-Timer nicht von z.B. der Firmware deaktiviert oder rekonfiguriert werden kann, wenn er gestartet wurde, wird er als cyber-resilient betrachtet. [17] Dies trifft auch auf solche Systeme zu, bei denen das Gerät kompromittiert wurde, aber der Watchdog-Timer nicht deaktiviert oder vom Gerät hinausgezögert werden kann. Es gibt verschiedene Arten an Watchdog-Timern [17]. Der Watchdog-Timer, der in dieser Arbeit betrachtet wird, kann nur von einem autorisierten Server mithilfe sogenannter *Deferraltickets* (siehe auch Kapitel 3) bedient werden.

Informationssysteme haben viele mögliche Angriffsflächen, die sie nicht ausnahmslos schützen können. Ein realistischeres Ziel ist in dem Fall die Cyber-Resilienz [20]. Jäger *et al.* [20] beschreiben Systeme oder Netzwerke, die kompromittierte Teile vom restlichen System isolieren oder einen vertrauenswürdigen Zustand wiederherstellen können, als resilient. Dabei sollen sie nur einen geringen Einfluss auf die Funktionalität des Gesamtsystems haben. Um diese Resets zur Wiederherstellung des Systems garantieren zu können, verwenden Jäger *et al.* [20] einen AWDT. Nach dem Reset kehrt das System in einen nicht-kompromittierten Zustand zurück. Besonders in der Industrie ist es wichtig, dass Systeme schnell wieder richtig funktionieren, da ansonsten hohe Kosten entstehen könnten [20].

2.2. Aktor-Framework Thespian

Das grundlegende Modell des Framewoks *Thespian*, das von der Hosting-Gruppe des Unternehmens GoDaddy entwickelt wurde, ist ein Aktorenmodell [49][50]. Die Open-Source Python-Bibliothek stellt nebenläufige, verteilte und fehlertolerante Anwendungen bereit. Das Design ist vereinfacht und es besteht die Möglichkeit, später ohne die Aktor-basierte Anwendung an sich zu ändern, die Gleichzeitigkeit oder den Transport anzupassen. [49] So können weitere Aktoren dynamisch hinzugefügt oder entfernt werden, wie es z.B. bei Cloud-Anwendungen der Fall sein kann.

2.2.1. Aktorenmodell

Das Aktorenmodell besteht aus Aktoren, die ausschließlich durch den Versand von Nachrichten miteinander kommunizieren. Je nach Art der empfangenen Nachricht führt ein Aktor eine bestimmte Aktion durch. So kann ein Programmcode in viele verschiedene Aktoren aufgeteilt werden, von dem jeder nur eine Aufgabe hat. Jeder Aktor hat eine Adresse, über die er Nachrichten empfangen kann. Um die Nachricht zu senden, wird die Adresse des Empfängers benötigt. Dieser wiederum benötigt eine Art Briefkasten, der nach dem FIFO-Prinzip arbeitet. [51]

2.2.2. Umsetzung im Framework

Das Aktorensystem, das eine externe Komponente darstellt, wird vom Thespian-Framework bereitgestellt und verwaltet. Ein Aktorensystem kann von einem externen Sender, z.B. von einem Python-Skript, erstellt werden und von diesem eine Nachricht empfangen. [52]

Ein Aktorensystem kann folgende Funktionen ausführen [49]:

- Erstellen von Aktor-Instanzen
- Verwalten der Nachrichtenverteilung von einem Aktor zu einem anderen Aktor
- Verwalten der Lebenszyklen der Aktoren
- Planung und Verwaltung der Nebenläufigkeit und des Zeitmanagements zwischen den Aktoren

Um einen Aktor auf einem System ausführen zu können, muss dort eine Instanz von Thespian gestartet werden. Die in einem Aktorensystem laufenden Aktoren sind unabhängig voneinander und können als eigene Prozesse oder Threads laufen. Da sie nicht voneinander abhängen, können sie auch auf unterschiedlichen Systemen befinden. Thespian verwaltet die Kommunikation zwischen den Aktoren selbst. [49]

Aufgrund der Unabhängigkeit der Aktoren von anderen Aktoren, behalten diese ihren internen Zustand bei und teilen diesen Zustand und ihren Speicher nicht mit anderen Aktoren.

Aktoren können verschiedene Aktionen ausführen, wenn sie eine Nachricht erhalten [49]:

1. Senden einer endlichen Anzahl an Nachrichten
2. Erstellen einer endlichen Anzahl an weiteren Aktoren
3. Aktualisieren des internen Zustands

Damit ein Aktor einem anderen Aktor eine Nachricht senden kann, benötigt er die Adresse des Empfänger-Aktors. Die Adresse kann entweder in einer Nachricht enthalten sein, die er selbst erhalten hat oder er erhält sie durch das direkte Erstellen des Aktors, mithilfe des Rückgabewerts von `self.createActor()`, an den die Nachricht gesendet wird. Eine Nachricht wird mit `self.send(ActorAddress, message)` an den empfangenden Aktor gesendet. [52]

Das Verhalten der Aktoren ist bei der Verwendung der `simpleSystemBase` deterministisch. Da in der Arbeit aber gewollt ist, dass mehrere Aktoren parallel und verteilt ausgeführt werden, ist die Nutzung von `simpleSystemBase` nicht möglich. [53] Stattdessen wird die `multiprocTCPBase` verwendet, die die nebenläufige Ausführung der Aktoren unterstützt. Bei der nebenläufigen Ausführung kann keine bestimmte Reihenfolge bei den empfangenden Nachrichten garantiert werden.

Ein Aktor kann mit der `receiveMessage`-Funktion Nachrichten unterschiedlicher Typen empfangen und anschließend diese Nachrichten je nach Typ verarbeiten. Eine `WakeupMessage` ist ein Nachrichtentyp, den ein Aktor durch eine eigene `receiveMessage`-Funktion abfangen kann. Wenn der Aktor eine solche Nachricht erhält, führt er eine festgelegte Aktion aus. Diese `WakeupMessage` wird immer in einem festgelegten Zeitabstand an den gewünschten Aktor gesendet. Ein Aktor kann auch sich selbst eine `WakeupMessage` senden (u.a. Abschnitt 4.5). Weitere Nachrichtentypen in Bezug auf diese Arbeit werden im Implementierungskapitel 4 näher erläutert, sobald sie erwähnt werden.

2.3. Kryptografie

Kryptografie ist ein Teilbereich der Kryptologie, bei dem es sich um den Entwurf verschiedener Verschlüsselungsverfahren handelt [48]. Das Ziel der Kryptografie ist es Nachrichten, die von einer Person - Alice - zu einer anderen Person - Bob - gesendet werden, so zu verschlüsseln und entschlüsseln, dass nur diese beiden Personen die Nachricht im Klartext lesen können. Um diese Verschlüsselung zu erreichen, werden *kryptografische Protokolle* verwendet. Ein Protokoll ist eine Festlegung von Verfahren und Regeln, an die sich z.B. beim Nachrichtenaustausch gehalten werden muss. Ein Beispiel für ein kryptografisches Protokoll ist die Verwendung eines einfachen Verschlüsselungsalgorithmus. Protokolle können sich auch aus vielen kleineren Protokollen zusammensetzen.

Kryptografische Primitive - wie z.B. Verschlüsselungsalgorithmen, Schlüsselherleitungen oder der Schlüsselaustausch - basieren größtenteils auf mathematischen Problemen, die unter Verwendung klassischer Computer nicht gelöst werden können, da die Berechnung zu lange dauern würde.

2.3.1. Klassische Kryptografie

Bei der klassischen Kryptografie, die sich in symmetrische und asymmetrische Kryptografie aufteilt, handelt es sich um Verschlüsselungsverfahren, die von klassischen Computern nicht in polynomieller Zeit gebrochen werden können. In den nachfolgenden Abschnitten werden die asymmetrische Kryptografie (Abschnitt 2.3.1.1), Einwegfunktionen (Abschnitt 2.3.1.2), digitale Signaturen (Abschnitt 2.3.1.3), kryptografische Hashfunktionen (Abschnitt 2.3.2) und elliptische Kurven (Abschnitt 2.3.1.4) näher erläutert. Abschließend wird die Sicherheit von Schlüssellängen (Abschnitt 2.3.3) sowie mögliche Angriffe (Abschnitt 2.3.4) beschrieben.

2.3.1.1. Asymmetrische Kryptografie

Die asymmetrische Kryptografie wird auch Public-Key-Kryptografie genannt. Hierbei handelt es sich um einen Nachrichtenaustausch zwischen zwei Parteien, bei dem jeder Kommunikationspartner ein Schlüsselpaar - bestehend aus einem öffentlichen und einem privaten Schlüssel - besitzt. Der öffentliche Schlüssel kann über einen unsicheren Kanal an den jeweiligen Kommunikationspartner gesendet werden. Um zu kommunizieren, verschlüsselt Alice eine Nachricht mit dem öffentlichen Schlüssel von Bob mit $c = E(m)_B$ und sendet sie an Bob. Bob erhält die Nachricht und entschlüsselt sie mit seinem privaten Schlüssel $m = D(c)_b$. Der Angreifer - kann zwar die Nachricht abfangen, aber sie nicht entschlüsseln, da er dazu den privaten Schlüssel von Bob benötigt. Die Gefahr hierbei ist das Abfangen der Schlüssel durch eine dritte Person, die sich dann als Alice und Bob ausgibt und mit diesen beiden kommuniziert (vgl. Abschnitt 2.3.4). Bei der symmetrischen Kryptografie besitzen Alice und Bob einen gemeinsamen Schlüssel zum Austausch von Nachrichten. Verschlüsselungs- und Entschlüsselungsverfahren sind hierbei sehr ähnlich [54].

Laut Paar [54] sind asymmetrische Verfahren rechenintensiv und für die Verschlüsselung von langen Nachrichten nicht geeignet. Daher wird die Public-Key-Kryptografie meistens zum Schlüsselaustausch von symmetrischen Schlüsseln verwendet. In dieser Arbeit wird das Thema *Schlüsselaustausch* nicht behandelt.

2.3.1.2. Einwegfunktionen

Bei asymmetrischen Verschlüsselungsverfahren (Abschnitt 2.3.1.1) werden Einwegfunktionen verwendet. Einwegfunktion bedeutet, dass die Berechnung in eine Richtung leicht durchzuführen ist, aber in die andere unter Verwendung klassischer Computer bis zu 100.000 Jahre dauern würde. Diese Einwegfunktionen basieren hauptsächlich auf zwei mathematischen Problemen - dem diskreten Logarithmus-Problem (z.B. Diffie-Hellman Key Exchange (DHKE)) sowie dem Faktorisierungsproblem (z.B. Rivest-Shamir-Adleman-Algorithmus (RSA)) [54]. Beim Faktorisierungsproblem wird davon ausgegangen, dass das Produkt zweier großer Primzahlen schnell zu berechnen ist. Wenn allerdings nur das Produkt gegeben ist, ist es nicht möglich daraus die verwendeten Produzenten zu berechnen. Das DLP ist ähnlich schwer zu berechnen [54].

2.3.1.3. Digitale Signaturen

Eine digitale Signatur kann nur mithilfe eines Public-Key-Kryptosystems durchgeführt werden, da sowohl der öffentliche als auch der private Schlüssel der Kommunikationspartner benötigt wird. Beim Erstellen der Signatur wird eine Nachricht mit einer kryptografischer Hashfunktion gehasht. Anschließend wird dieser Hash als Nachricht angesehen und mit dem privaten Schlüssel des Senders signiert, wodurch eine Signatur erstellt wird. Der Empfänger verifiziert die Signatur, indem er die Nachricht mit dem öffentlichen Schlüssel des Senders verifiziert und anschließend den dadurch erhaltenen Hash mit dem selbst generierten Hash der mitgesendeten originalen Nachricht vergleicht. Ist dies der Fall, so wird die Nachricht als authentisch angesehen. [55]

2.3.1.4. Elliptische Kurven Kryptografie (ECC)

Elliptische Kurven sind eine weitere Algorithmenfamilie, die zur asymmetrischen Kryptografie zählt. Ein Vorteil dieser Algorithmen ist, dass sie die gleiche Sicherheit wie RSA oder DL-Verfahren bieten, aber kürzere Schlüssel und Signaturen benötigen, sowie häufig schneller sind und eine geringere Bandbreite benötigen [54]. Weiterhin können RSA-Signaturen mit geringen Exponenten bei der Verwendung von ECC jedoch schneller verifiziert werden.

Bei ECC handelt es sich um eine Verallgemeinerung der auf dem DLP basierenden Verfahren über endlichen Körpern. Dies ist der Grund, warum z. B. DHKE (basierend auf dem DLP) auch mit elliptischen Kurven realisiert werden kann. [54] Die zugrundeliegende Mathematik ist um einiges komplexer als bei RSA und dem DLP. Die Grundlagen dazu können u.a. in [54] nachgelesen werden. Beim *Elliptic Curve Digital Signature Algorithm (ECDSA)* handelt es sich um einen digitalen Signaturalgorithmus basierend auf elliptischen Kurven. Es handelt sich hierbei um eine Variante der digitalen Signatur nach ElGamal, die mit elliptischen Kurven verwendet werden kann.

2.3.2. Kryptografische Hashfunktionen

Eine Hashfunktion ist eine Funktion h, die als Eingabe x eine Nachricht beliebiger Länge erwartet und diese auf Ausgaben einer festgelegten Länge abbildet [55] (siehe Gleichung 2.1). Wird an der Nachricht nun auch nur ein einziger Buchstabe verändert und erneut ein Hashwert berechnet, so kommt ein um viele Stellen verschiedener Hashwert heraus als der zuvor berechnete. [55] So kann festgestellt werden, ob eine Nachricht bei der Übertragung verändert wurde. [55] Diese Eigenschaft wird Integrität genannt und ist eine der Hauptschutzziele der IT-Sicherheit (siehe Abschnitt 2.1.1).

$$Hashwert = h(x) \tag{2.1}$$

Die Berechnung des Hashwerts erfolgt in polynomieller Zeit [55]. Da Hashfunktionen Einwegfunktionen sind, ist es in polynomieller Zeit nicht möglich eine Rückberechnung des Hashwerts auf die ursprüngliche Nachricht durchzuführen.

Ein Beispiel für Gruppen standardisierter kryptografischer Hashfunktionen sind der Secure Hash Algorithm (SHA) und SHAKE. *SHA* hat drei verschiedene Hashfamilien (SHA1, SHA2, SHA3), von denen in dieser Arbeit ausschließlich mit den SHA2-Algorithmen gearbeitet wird. Die verschiedenen SHA-Algorithmen unterscheiden sich laut [55] in der Wahl der verwendeten Konstanten und Kompressionsfunktionen. *SHAKE* ist eine *extendable-output function (XOF)* der SHA3-Familie und der Name setzt sich zusammen aus **SHA** und **KE**ccak. SHAKE ist *light-weight* und energieeffizient und kann somit gut in IoT-Geräten verwendet werden [56]. Die Ausgabelänge der Hashfunktion ist bei SHAKE variabel. Die Sicherheit der Algorithmen ist anhand der Suffixe 128 oder 256 festgemacht. Wie genau SHAKE funktioniert, wird in dieser Arbeit nicht näher betrachtet. Weiterführende Informationen sind in [57] zu finden.

2.3.3. Sicherheit von Schlüsseln und Hashes

Schlüssellänge	Sicherheit (klassische Computer)
56-64 Bit	Kurzfristig: einige Stunden oder Tage
112-128 Bit	Langfristig: einige Jahrzehnte
256 Bit	Langfristig: einige Jahrzehnte

Tabelle 2.2.: Abschätzung der Zeit, die ein Angreifer benötigt, um den Schlüssel von symmetrischen Verfahren zu berechnen. Übernommen aus [54].

In der Kryptografie hat die Bitsicherheit verschiedener Schlüssellängen eine bedeutende Rolle. Daher werden in Tabelle 2.2 mögliche Schlüssellängen in Bit mit der Zeitspanne aufgeführt, in der diese sicher sind. Dies wird sowohl für klassische Computer als auch für Quantencomputer in [54] von Paar geschätzt.

Eine Bitsicherheit von z.B. 80 Bit bedeutet, dass der beste Angriff 2^{80} Schritte benötigt, um den Schlüssel zu berechnen. Bei symmetrischen Algorithmen ist die Schlüssellänge n in Bit gleich der Bitsicherheit. Bei asymmetrischen Algorithmen werden z.T. viel längere Schlüssel benötigt, um das gleiche Sicherheitsniveau zu erhalten. Um zu gewährleisten, dass der Schlüssel nicht bestimmt werden kann, sollte mindestens eine Bitsicherheit von 128 Bit erreicht werden. Die größeren Schlüssellängen machen den Algorithmus rechenintensiver, wodurch sich der Rechenaufwand kubisch erhöht.

Bei einer kryptografischen Hashfunktion geht es nicht um die Bitsicherheit, sondern um die Kollisionsresistenz. Kollision beschreibt die Möglichkeit, dass durch Brute Force zwei gleiche Hashwerte aus zwei verschiedenen Eingaben generiert werden können.

Laut [54] ist die Langzeitsicherheit von 112 bis 128 Bit für einige Jahrzehnte ausreichend, außer der Angreifer besitzt einen Quantencomputer mit genügend Qubits. Wenn eine Sicherheit von 256 Bit für Schlüssel und Hashwerte vorliegt, so sind diese auch gegen Angriffe von Quantencomputern geschützt [54]. Dies gilt jedoch nur für symmetrische Algorithmen, bei denen Quantencomputer die effektive Länge der Schlüssel halbieren. Asymmetrische Algorithmen können gegenüber Quantenalgorithmen grundsätzlich nicht mehr als sicher

angenommen werden [54]. Daher wird im folgenden Verlauf dieser Arbeit auf dieser Aussage basierend angenommen, dass eine Bitsicherheit von mindestens 256 Bit gewährleistet werden muss, damit die symmetrischen Algorithmen als quantensicher angenommen werden können.

2.3.4. Angriffe

Die Datensicherheit von Informationssystemen ist durch Man-in-the-Middle-Angriffe, passives Abhören, aktives Mithören, Replay-Angriffe und vielem mehr bedroht [58] (vgl. Abschnitt 2.3.4). In dieser Arbeit wird auf Datensicherheit mithilfe von Kryptografie und Sicherheitsprotokollen eingegangen.

Es wird von einem Angriff gesprochen, wenn eine dritte Partei - der Angreifer - die Nachricht auf dem Transportweg zwischen Alice und Bob abfängt und sie mitliest oder verändert. In den verwendeten Beispielen kann Bob der Sender oder der Empfänger sein und Alice der Empfänger respektive Sender. Es gibt verschiedene Arten von Angriffen auf eine Kommunikation.

Bei passiven Angriffen hört der Angreifer die Konversation mit oder sammelt die gesendeten Nachrichten, ohne sie zu verändern [54]. Diese Art von Angriffen kann auch jetzt schon bei der Kommunikation zwischen einem IoT-Gerät und einem Server ausgenutzt werden. Wie in der Einleitung (Abschnitt 1.1) beschrieben, gibt es das Sprichwort *Store now, decrypt later*. Es wird davon ausgegangen, dass bestimmte Organisationen, die heute versendeten Nachrichten sammeln und darauf warten, dass Quantencomputer diese Nachrichten in Zukunft entschlüsseln können [7].

Bei aktiven Angriffen wird angenommen, dass der Angreifer die gesendeten Nachrichten abgreift, sie verändert oder neu erstellt und erst dann an den eigentlichen Empfänger sendet [54]. Ein Beispiel dafür ist der Man-in-the-middle-Angriff, bei dem der Angreifer die öffentlichen Schlüssel der Kommunikationspartner z.B. beim Schlüsselaustausch aufgreift und Alice und Bob jeweils seinen eigenen öffentlichen Schlüssel sendet [54].

Ein weiteres Beispiel eines aktiven Angriffs ist der Replay-Angriff, bei dem ein Angreifer zuvor versendete Daten sammelt und zu einem späterenZeitpunkt gezielt wieder in die Kommunikation einspeist [59]. Hierbei gibt der Angreifer vor Alice zu sein und Bob eine Nachricht zu senden. Beispielsweise könnte dies bei einer Überweisung von Geld passieren.

Alice und Bob gehen in beiden genannten Fällen davon aus, dass sie sich miteinander unterhalten, obwohl sie beide mit dem Angreifer kommunizieren.

Eine Maßnahme um gegen Replay-Angriffe vorzugehen ist die Verwendung von sogenannten *Noncen*. Der Begriff Nonce ist eine Abkürzung für *used only once* oder *number used once*. Diese Zahl darf nicht mehr als einmal verwendet werden und dient zur Randomisierung des Chiffretexts. Wird angenommen, dass zweimal genau die gleiche Nachricht versendet wird, aber jeweils eine unterschiedliche Nonce mit dem geheimen Schlüssel verwendet wird, so sind zwei verschiedene Chiffretexte das Ergebnis für die gleiche Nachricht.

2.4. Post-Quantum-Kryptographie

In diesem Abschnitt wird eine Einführung in die Post-Quantum-Kryptographie (PQC) gegeben. Dazu werden verschiedene Klassen von PQC-Algorithmen (Abschnitt 2.4.1) und quantensichere Standards (Abschnitt 2.4.2) vorgestellt.

Die klassische Kryptografie wie sie noch heute verwendet wird, besteht (wie in Abschnitt 2.3 beschrieben) aus symmetrischer und asymmetrischer Kryptografie. Die asymmetrischen Verfahren (siehe Abschnitt 2.3.1.1) basieren auf den in Abschnitt 2.3.1.2 beschriebenen Einwegfunktionen. Zur Public-Key-Kryptografie zählen außerdem die RSA-Signaturen, die auf der Schwierigkeit des Faktorisierungsproblems basieren. Des Weiteren gehört der Diffie-Hellman Schlüsselaustausch, der auf der Schwierigkeit des Problems elliptischer Kurven Diskreten Algorithmen basiert, zu den asymmetrischen Verfahren.

Shor hat 1994 einen Algorithmus veröffentlicht, der die beiden genannten Probleme in effizienter Zeit auf einem Quantencomputer lösen kann. Die symmetrische Kryptografie zu der AES und die kryptografischen Hashfunktionen wie die SHA-2-Algorithmen gehören, sind auch gefährdet. Mithilfe von Grover's Algorithmus, der 1996 veröffentlicht wurde, kann mit einem Quantencomputer ein Schlüssel in der Hälfte der Zeit gefunden werden [7]. Bei Hashfunktionen muss die Ausgabegröße des Hashes länger sein, damit es schwieriger wird, durch Zufall den gleichen Hashwert zu errechnen. Dies wird auch Kollisionsresistenz genannt. [54] Durch die Erhöhung der Schlüsselgrößen und Hashlängen kann diesem Problem entgegengewirkt werden. Wie in Tabelle 2.2 bereits aufgeführt, sollten langfristig mindestens Schlüssellängen von 256 Bit verwendet werden, dies gilt auch für Hashlängen. Für diese dauert die Berechnung mithilfe eines Quantencomputers bis zu einige Jahrzehnte. [54]

Zur Berechnung des Elliptic Curve Discrete Logarithm Problem (ECDLP) mit 256 Bit werden 2330 Qubits benötigt und zum Lösen des Faktorisierungsproblems (RSA) für 3072 Bit werden 6146 Qubits benötigt [60]. IBM veröffentlichte im November 2022 den Quantenprozessor Osprey mit 433 Qubits [61].

Des Weiteren empfiehlt das BSI hybride Verschlüsselungsverfahren zu benutzen [7]. Das bedeutet, dass neben den aktuell implementierten klassischen Verfahren wie z.B. RSA noch ein weiterer quantensicherer Algorithmus parallel dazu implementiert wird. Um die Algorithmen in einer Implementierung zu brechen, müssen dafür beide Verfahren, sowohl der klassische als auch der quantensichere Algorithmus gebrochen werden. Dies ist vorallem jetzt in der Transition von klassischer zu quantensicherer Kryptografie wichtig. Grund dafür ist, dass die quantensicheren Algorithmen noch nicht ausgiebig genug getestet wurden und es möglich ist, dass plötzlich eines dieser quantensicheren Verfahren gebrochen wird. Sollte dies bei einer hybriden Implementierung der Fall sein, ist die Implementierung jedoch weiterhin mit dem klassischen Algorithmus geschützt - so fern dieser nicht auch gebrochen wurde. [7]

2.4.1. Verschiedene Gruppen von PQC-Algorithmen

Für Public-Key-Verschlüsselung und digitale Signaturen gibt es PQC-Algorithmen, wovon es fünf verschiedene Arten gibt. Diese lauten:

- Gitterbasierte Verfahren
- Hashbasierte Verfahren
- Codebasierte Verfahren
- Isogeniebasierte Verfahren
- Multivariate Verfahren

Von diesen verschiedenen Arten werden in dieser Arbeit nur die gitterbasierten und die hashbasierten Verfahren kurz beschrieben, da die anderen drei bisher nicht als zukünftige quantensichere Standards festgelegt wurden (siehe Abschnitt 2.4.2)

Die gitterbasierte Kryptografie baut auf dem mathematisch schwer zu lösenden Shortest Vector Problem (SVP) in mathematischen Gittern auf [7] ist. Ein Gitter ist eine Sammlung aus Punkten in einem Raum. Dieser Raum wird als privater Schlüssel angesehen und eine verschlüsselte Version dieser Basis ist der öffentliche Schlüssel.

Bei den hashbasierten Verfahren liegt dem Algorithmus die Sicherheit eines digitalen Signaturverfahrens bzw. der dabei verwendeten Hashfunktion zugrunde und wird mithilfe dieser konstruiert [7]. Es wird unterschieden zwischen zustandslosen und zustandsbehafteten hashbasierten Signaturverfahren. Die Zustandsbehaftung geht auf sogenannte Merkle-Signaturen zurück, was bedeutet, dass der Ersteller der Signatur sich merken muss, welche Einmal-Signaturschlüssel er schon verwendet hat. Bei den zustandslosen Verfahren wie z.B. SPHINCS, das auf einer Konstruktion von Goldreich basiert, muss der Signaturersteller sich nicht mehr merken, welche Signaturschlüssel er schon verwendet hat. Dadurch entsteht der Nachteil einer längeren Signatur. [7]

2.4.2. Quantensichere Standards

Das NIST hat 2016 einen Aufruf zur Einreichung von PQC-Verfahren sowohl für Public-Key-Verfahren als auch für digitale Signaturen gestartet [12]. Dieser wird auch häufig als Wettbewerb bezeichnet, in dem mehrere Verfahren eingereicht werden, die getestet werden und am Ende standardisiert werden sollen.

2.4.2.1. Sicherheitsanforderungen

Das NIST hat fünf verschiedene Sicherheitsanforderungen für die zu standardisierenden Algorithmen gestellt, die aufsteigend in ihrer Stärke dargestellt werden [12]: *Jeder Angriff, der die einschlägige Sicherheitsdefinition umgeht, muss Berechnungsressourcen erfordern, die vergleichbar oder besser sind als die, die für die:*

1. *Schlüsselsuche bei einer Block-Chiffre mit einem 128-Bit-Schlüssel (z.B. AES128)*
2. *Kollisionssuche auf einer 256-Bit-Hashfunktion (z.B. SHA256/SHA3-256)*
3. *Schlüsselsuche bei einer Block-Chiffre mit einem 192-Bit-Schlüssel (z.B. AES192)*
4. *Kollisionssuche auf einer 384-Bit-Hashfunktion (z.B. SHA384/SHA3-384)*
5. *Schlüsselsuche bei einer Block-Chiffre mit einem 256-Bit-Schlüssel (z.B. AES256)*

benötigt werden.

2.4.2.2. Empfehlungen des BSI und des NIST

Das BSI hat im Jahr 2021 ein Dokument über die Einführung von quantensicherer Kryptografie herausgebracht [7]. In diesem Dokument werden Grundlagen erläutert, Entwicklungen dargelegt, sowie Handlungsempfehlungen für die Verwendung quantensicherer Kryptografie gegeben. Das BSI [7] spricht sich für die Anwendung hybrider Ansätze für Schlüsseleinigung und digitale Signaturen aus. Hybrid bedeutet in diesem Zusammenhang, dass sowohl klassische als auch quantensichere Algorithmen implementiert werden. Dies ist hauptsächlich für die Übergangszeit von der Verwendung klassischer zu quantensicherer Kryptografie zu empfehlen, da die quantensicheren Algorithmen noch nicht ausreichend getestet wurden und somit noch Mängel aufweisen. Wird der quantensichere Algorithmus gebrochen, so wird die Nachricht zumindest noch durch den klassischen Algorithmus geschützt. Andersherum ist dies auch möglich.

Das BSI hat Anfang 2020 erstmals zwei quantensichere Schlüsseleinigungsverfahren in einer technischen Richtlinie empfohlen: Des Weiteren empfiehlt das BSI den Einsatz von hashbasierten Signaturen, die allerdings nicht für jeden Anwendungsfall geeignet sind [7]. Die drei folgenden Signaturverfahren zur Authentisierung werden außerdem vom BSI empfohlen:

- CRYSTALS-Dilithium (gitterbasiert)
- FALCON (gitterbasiert)
- SPHINCS+ (zustandslos, hashbasiert)

Neben diesen drei werden auch Merkle-Signaturen als quantensichere Signaturverfahren empfohlen. Das BSI hat in absehbarer Zeit nicht vor multivariate oder isogeniebasierte Verfahren zu empfehlen. [7]

Das NIST hat im Juli 2022 das Ende der dritten Runde des PQC-Standardisierungsprozesses und damit die Finalisten CRYSTALS-Dilithium, FALCON und SPHINCS+ verkündet [62]. Hierbei wurden Algorithmen bekannt gegeben, die entweder standardisiert oder noch weiter getestet werden sollen. Des Weiteren wurde ein Aufruf gestartet, dass bis 01. Juni 2023 weitere *public-key (quantum-resistant) digital signature algorithms* eingereicht werden können, damit das Repertoire an digitalen Signaturverfahren noch erweitert werden kann.

2.4.2.3. CRYSTALS-Dilithium

Dieser Signaturalgorithmus ist Teil der **CRY**ptographic **S**ui**T**e for **A**lgebaric **L**attice**S** und wird im weiteren Verlauf der Arbeit *Dilithium* genannt. Dilithium ist ein gitterbasierter Signaturalgorithmus, der sicher gegenüber *Chosen Message Attacks* ist, die auf der Schwierigkeit zur Lösung des Gitterproblems über Modulgittern des **L**earning **W**ith **E**rror basieren. [63]

Die Designer des Verfahrens empfehlen Dilithium im hybriden Modus zu implementieren sowie das Parameterset 3 zu verwenden, das eine Bitsicherheit von mehr als 128 gegen alle klassischen Angriffe und Quantenangriffe aufzuweisen hat [64]. Des Weiteren gehen die Designer davon aus, dass Dilithium den kleinsten Größen, wenn es um die Kombination von öffentlichem Schlüssel und Signatur der gitterbasierten Signaturverfahren hat [64].

Dilithium verwendet SHAKE-128 und SHAKE-256. Es gibt eine Möglichkeit den Algorithmus deterministisch zu gestalten, in dem ein Seed an den privaten Schlüssel angehängt wird, um in Kombination mit der Nachricht eine Zufälligkeit zu erreichen. Die Referenz-Implementierung wurde auf einem Computer mit Intel Core-i7 6600U (Skylake) CPU und einer optimierten Implementierung unter der Verwendung von AVX2-Vektorinstruktionen für drei Parametersets getestet. Tabelle 3.2 in Abschnitt 3.2.2 zeigt eine Übersicht über diese Parametersets.

Um die Algorithmen bezüglich ihrer NIST-Sicherheitslevel vergleichen zu können, wird Dilithium2 bei den Messungen nicht berücksichtigt, da es keinen vergleichbaren Algorithmus von Falcon oder SPHINCS+ mit dem Sicherheitslevel 2 gibt.

2.4.2.4. FALCON

FALCON ist ein gitterbasierter kryptografischer Signaturalgorithmus, dessen Name sich aus dem Akronym **FA**st-Fourier **L**attice-based **CO**mpact Signatures over **N**TRU ergibt. NTRU ist ein ringbasiertes Public-Key-Kryptosystem [65]. FALCON wird in weiteren Verlauf der Arbeit *Falcon* genannt.

Laut der Webseite des Algorithmus hat Falcon folgende Eigenschaften [66]:

- *Sicherheit*: Durch die Verwendung eines Gaußschen Abtasters wird das Durchsickern von Informationen zum geheimen Schlüssel vernachlässigbar klein und es wird eine sehr hohe Anzahl (2^{64}) an Signaturen garantiert.
- *Kompaktheit*: Durch die Verwendung von NTRU-Gittern sind die Signaturen kürzer. Die Länge des öffentlichen Schlüssels ist in etwa gleich groß wie bei anderen gitterbasierten Signaturverfahren. Die Sicherheitsgarantien bleiben gleich.
- *Geschwindigkeit*: Durch die Verwendung der schnellen Fourier-Abtastung, können die gängigen klassischen Computer pro Sekunde Tausende Signaturen erstellen. Deren Verifizierung ist 5- bis 10-Mal schneller.

- *Skalierbarkeit*: Die Verwendung langfristiger Sicherheitsparameter kann zu moderaten Kosten ermöglicht werden. Operationen haben O(n log n)
- *Speichereffizienz (RAM)*: Durch den verbesserten Algorithmus der Schlüsselgenerierung benötigt Falcon weniger als 30 KB des Random Access Memory.

Es gibt zwei Varianten des Falcon-Algorithmus: Falcon-512 und Falcon-1024.

2.4.2.5. SPHINCS+

SPHINCS+ ist ein zustandsloses, hashbasiertes Signaturverfahren, das vom NIST standardisiert werden soll. Es wurden drei verschiedene Signaturverfahren eingereicht:

- SPHINCS+ - SHAKE256
- SPHINCS+ - SHA256
- SPHINCS+ - Haraka

Diese drei Signaturverfahren bestehen jeweils aus verschiedenen SPHINCS+-Varianten (siehe Tabelle 2.3). Die Designer dieses Verfahrens veröffentlichen Beispiel-Parametersets in der Spezifikation [67], die die Sicherheitslevel 1, 3 und 5 des NIST PQC Wettbewerbs betreffen. Diese Parametersets wurden für die dritte Runde des Wettbewerbs geändert [67]. Die SPHINCS+-Varianten stellen für die drei NIST-Sicherheitslevel jeweils vier Parametersets bereit: *s-simple* und *s-robust* sowie *f-simple* und *f-robust*. Die Varianten sollen verschiedene Trade-Offs zwischen Signaturgröße und Geschwindigkeit bereitstellen. *s* steht für *small* und ist ein speicherplatzoptimiertes Parameterset. *f* steht für *fast* und ist ein geschwindigkeitsoptimiertes Parameterset. Die Variante *simple* wurde in der Abgabe zur zweiten Runde des NIST-Wettbewerbs eingeführt und verwendet, im Gegensatz zu der bestehenden Variante, die von dem Zeitpunkt an *robust* genannt wird, keine Bitmasken. [67]

Die Hasfunktionen SHAKE256 und SHA256 sind vom NIST standardisiert. Haraka ist keine vom NIST genehmigte Hashfunktion, daher wird in dieser Arbeit nicht weiter auf diese Variante eingegangen.

Variante	128s	128f	192s	192f	256s	256f
Sicherheitslevel (NIST)	1	1	3	3	5	5
Bitsicherheit	133	128	193	194	255	255
Signaturgröße (Bytes)	7.856	17.088	16.224	35.664	29.792	49.856

Tabelle 2.3.: Übersicht über die Sicherheitslevel, Bitsicherheit sowie die Signaturgrößen der verschiedenen SPHINCS+-Parametersets [67] für SHA256, SHAKE256 und Haraka. *s* steht für *space* (speicherplatzoptimiert) und *f* steht für *fast* (geschwindigkeitsoptimiert) [67].

Vorteile von SPHINCS+ sind laut den Designern [67] die kleinen Schlüsselgrößen und die einfache Analyse von State-of-the-art-Angriffen. Bei SPHINCS+ werden etablierte Bausteine wiederverwendet und der Algorithmus überlappt sich mit Extended Merkle Signature Scheme (XMSS). Die Sicherheit von SPHINCS+ basiert auf der Sicherheit der verwendeten Hashfunktion.

Die Nachteile von SPHINCS+ liegen laut den Designern [67] entweder in der langsamen Signaturerstellung oder in den großen Signaturen. Durch die Parametersets kann sich für eine der den beiden Varianten entschieden werden, solange der andere Nachteil tolerierbar ist [67].

3. Konzeption des Watchdog-Timer-Protokolls

In diesem Kapitel wird ein zunächst auf klassischer Kryptografie basierendes Watchdog-Timer-Protokoll neu konzipiert (Abschnitt 3.1). Anschließend werden drei verschiedene quantensichere Signaturverfahren in einer Übersicht mit den jeweiligen Eigenschaften dargestellt (Abschnitt 3.2), die die klassischen Algorithmen ersetzen könnten.
Dieses Kapitel dient als Grundlage für die Proof-of-Concept-Implementierung eines quantensicheren Watchdog-Timer-Protokolls in Kapitel 4.

3.1. Klassisches Watchdog-Timer-Protokoll

In diesem Abschnitt wird ein neues Watchdog-Timer-Protokoll konzipiert, das auf den Grundlagen und Abläufen der Systeme CIDER- [3] und Lazarus [2] basiert. Die Abgrenzungen zu den beiden bestehenden Protokollen werden in Abschnitt 3.1.1 zusammengefasst. In Abschnitt 3.1.2 wird der grobe Ablauf des Protokolls mithilfe eines Ablaufdiagramms (Abbildung 3.1) beschrieben und anschließend in kleinere Szenarien aufgeteilt (Abschnitt 3.1.4). Des Weiteren werden die einzelnen zu versendenden Nachrichten mit ihren Schutzzielen aufgeführt (Abschnitt 3.1.5) und die verwendeten kryptografischen Primitiven und Algorithmen vorgestellt (Abschnitt 3.1.6). In Abschnitt 3.1.7 werden die Konzepte und kryptografischen Primitiven aufgeführt, mit denen das neu konzipierte Watchdog-Timer-Protokoll umgesetzt wird.

3.1.1. Abgrenzung zu CIDER und Lazarus

In dieser Arbeit wird der Fokus auf die Ersetzung der klassischen Kryptografie im Watchdog-Timer-Protokoll durch quantensichere Kryptografie gelegt. Daher kann das neu konzipierte Protokoll als eine abgewandelte Version des CIDER- [3] bzw. Lazarus-Protokolls [2] angesehen werden. Es ist eine in Teilen vereinfachte Version der bestehenden Prokolle, da einige Systemkomponenten als gegeben angenommen werden oder in der späteren Implementierung (Kapitel 4) nicht betrachtet werden, da sie für die Beantwortung der Forschungsfragen nicht relevant sind.

Huber *et al.* [2] setzen eine Trennung von vertrauenswürdiger und nicht-vertrauenswürdiger Software bzw. Ausführungsumgebungen mit einer Trusted Execution Environment (TEE)

um. Innerhalb dieser TEE werden sicherheitskritische Funktionen ausgeführt, wohingegen außerhalb der TEE die nicht-vertrauenswürdige Software wie der Update-Downloader und die Business-Logik ausgeführt werden. Des Weiteren wird DICE und die Erweiterung DICE++ verwendet, um eine Attestierung der Kommunikationspartner durchzuführen [2]. Auf die Schlüssel wird, nachdem sie gelatchet wurden, nur noch lesend und nur von der vertrauenswürdigen Software in der TEE zugegriffen. Um das Konzept und die Implementierung dieser Arbeit so einfach wie möglich zu gestalten, wird keine sichere Ausführungsumgebung hergestellt und vom Einsatz von DICE abgesehen, da durch den Lazarus-Prototyp [35] bereits gezeigt wurde, dass dies möglich ist. Es wird davon ausgegangen, dass dies auch für quantensichere Algorithmen möglich ist.

Zur Vereinfachung wird bei der neuen Konzeption angenommen, dass das System nur aus einem Gerät und einem Server besteht. Um zu zeigen, dass ein Austausch der klassischen kryptografischen Primitive gegen quantensichere Primitive grundsätzlich möglich ist, wird sich auf die Erstellung von einem Schlüsselpaar pro Gerät und pro Server beschränkt. Mithilfe der beiden Schlüsselpaare können das Gerät und der Server sicher miteinander kommunizieren.
Bei Lazarus [2] werden noch zusätzliche Schlüssel erstellt, um z.B. die Attestierung eines Geräts mithilfe von DICE und DICE++ auch nach einem Update zu ermöglichen. Die Erstellung weiterer Schlüsselpaare liegt jedoch außerhalb des Rahmens dieser Arbeit.

3.1.2. Ablauf des Protokolls

Zur Übersicht über die zusammenhängenden Abläufe im Watchdog-Timer-Protokoll wird das Diagramm in Abbildung 3.1 teilweise aus dem Protokollablauf von Lazarus [2] übernommen, erweitert und teilweise abgewandelt. Mithilfe des Ablaufdiagramms in Abbildung 3.1 werden die möglichen Szenarien, die das Protokoll ablaufen kann, aufgezeigt. Im linken Bereich werden die Abläufe abgebildet, die innerhalb des Geräts möglich sind. Der rechte Bereich der Abbildung 3.1 stellt die Aufgaben des Servers dar.

Die Hauptfunktionalität des Protokolls ist der Watchdog-Timer im Gerät. Dieser hat wie bei CIDER [3] und Lazarus [2] beschrieben drei Schnittstellen zur Außenwelt:

- *AWDT_Init*
- *AWDT_GetNonce*
- *AWDT_PutTicket*

Mit der Schnittstelle *AWDT_Init* wird der Timer initialisiert und gestartet. Dabei erhält der Timer eine Zeitangabe, die ursprünglich vom Server kommt oder fest im Gerätespeicher hinterlegt ist. Diese Zeitangabe stellt eine Zeit dar, die vom Starten des Timers bis zum Ablauf des Timers verstreichen darf. Nach dem Ablaufen des Timers wird das Gerät neu gestartet, um das Protokoll resilient zu machen.

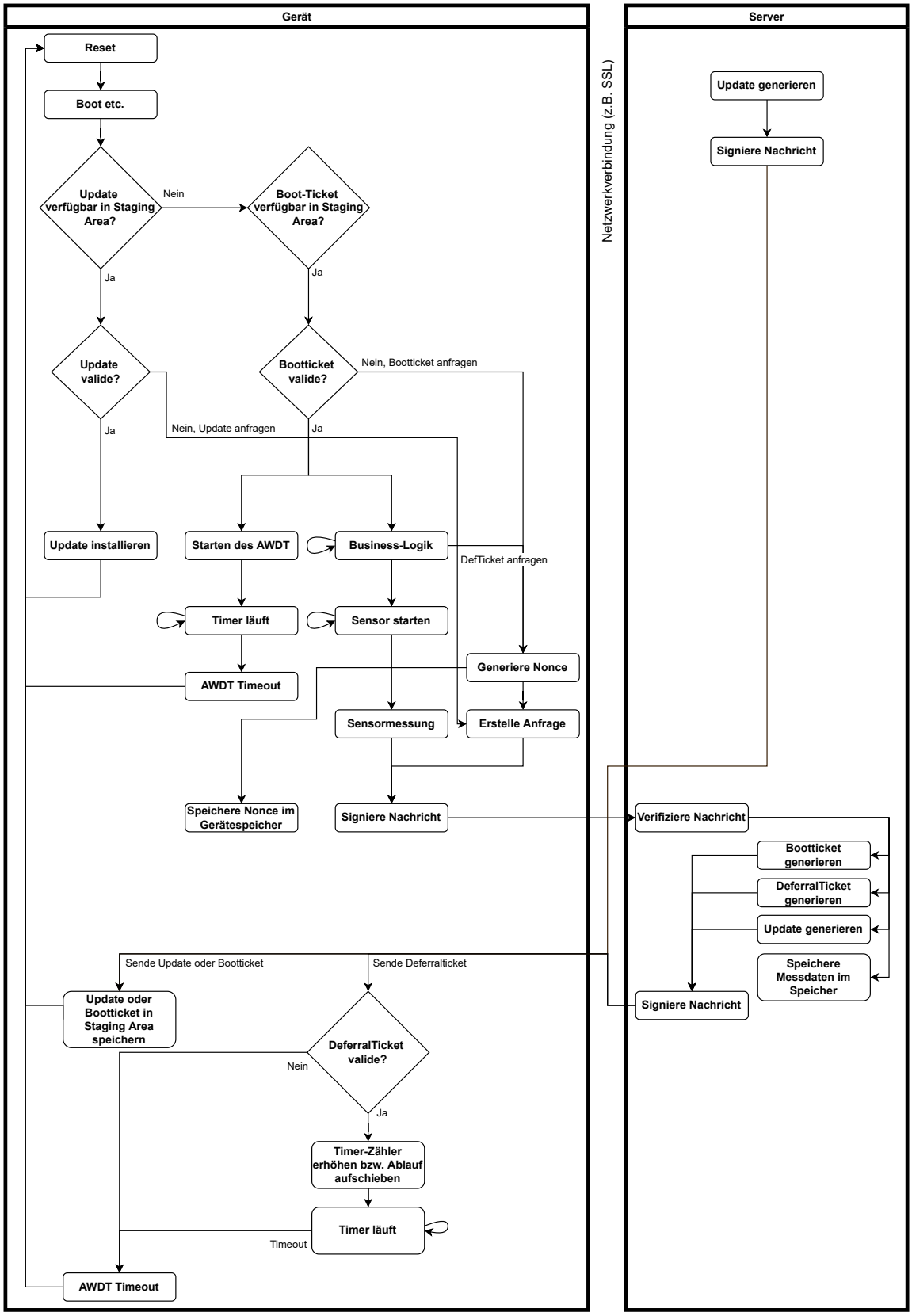

Abbildung 3.1.: Die Abläufe im Watchdog-Timer-Protokoll. Angelehnt an [2].

In bestimmten Zeitintervallen wird überprüft, ob der Timer bereits abgelaufen ist. Das Überprüfen wird durch einen auf sich selbstzeigenden Pfeil des Blocks *Timer läuft* dargestellt (vgl. Abbildung 3.1). Um ein Timeout zeitlich hinauszuzögern, generiert das Gerät eine Nonce mit der Schnittstelle *AWDT_GetNonce* und sendet diese an den Server. Der Server nutzt die Nonce und erstellt ein sogenanntes Deferralticket, das signiert und zurück an das Gerät gesendet wird. Die dritte Schnittstelle stellt das Anwenden des Deferraltickets mit *AWDT_PutTicket* dar. Der Timer erhält das Deferralticket und setzt den Timer auf die Zeit, die im Deferralticket steht. Das Gerät benötigt diese Deferraltickets vom Server damit es sich nicht aufgrund eines Timeouts herunterfahren muss (vgl. Abschnitt 2.1.2).

Grundsätzlich wird davon ausgegangen, dass das Gerät ausschließlich sich selbst neu starten und booten kann. Alles Weitere geschieht aufgrund bestimmter Zustände des Geräts, dass z. B. ein Update oder ein Bootticket in der sogenannten Staging Area, einem persistenten Speicher des Geräts, vorliegt. Der Server kann ein Update an das Gerät senden, wohingegen ein Bootticket vom Gerät beim Server angefragt werden muss. Das vom Server an das Gerät gesendete Update oder Bootticket wird in der Staging Area des Geräts gespeichert.

Ein Neustart kann durch ein Timeout des Watchdog-Timers oder ein erfolgreiches Abspeichern von einem Update oder einem Bootticket in der Staging Area des Geräts hervorgerufen werden. Nach einem erneuten Bootvorgang kann ein in der Staging Area befindliches Update oder Booticket verifiziert werden. Es wird immer zuerst überprüft, ob ein Update in der Staging Area verfügbar ist (vgl. Abbildung 3.1) und danach, ob ein Bootticket verfügbar ist. Ein Update ist somit relevanter als ein Bootticket, da das Update z.B. einen Sicherheitspatch für die Business-Logik oder die Firmware des Geräts beinhalten könnte. Ein valides Bootticket dient dazu, dass nach dem Bootprozess direkt in die Business-Logik gestartet werden kann, ohne dass der Server erneut kontaktiert werden muss. Das Bootticket ist sozusagen ein Freiticket, damit das Gerät mit der Ausführung der Funktionalität beginnen kann.

Die Business-Logik kann unterschiedliche Funktionen ausführen, wie z.B. das Messen der Temperatur mithilfe eines Sensors. Die gemessenen Daten werden zur Speicherung und Verarbeitung an den Server gesendet und in dieser Arbeit nicht weiter betrachtet. Der Sensor soll in einem bestimmten Zeitintervall einen Wert messen, dies wird durch den auf sich selbstzeigenden Pfeil dargestellt.

Der auf sich selbstzeigende Pfeil bei dem Kästchen *Business-Logik* stellt dar, dass die Business-Logik des Geräts in festgelegten Zeitintervallen ein Deferralticket beim Server anfragt. Ist das Gerät kompromittiert und kann deswegen keine Deferralticket-Anfragen mehr stellen, so wird es heruntergefahren und beim Neustart wieder in einen sicheren, nicht kompromittierten Zustand versetzt. Eine andere Möglichkeit ist, dass der Server durch z.B. Maßnahmen des Security Information and Event Management (SIEM) Kenntnis davon erhalten hat, dass das Gerät kompromittiert ist und kein Deferralticket mehr an das Gerät sendet, obwohl er eine Anfrage erhalten hat.

Ein vertrauenswürdiger und sicherer Zustand des Geräts kann durch das Update eines Servers oder durch einen Reset des Geräts wiederhergestellt werden. Mithilfe eines Updates vom Server kann die Sicherheitslücke, die zur Kompromittierung geführt hat, geschlossen

werden. Updates können vom Server gesendet werden, wenn z.B. der Händler des Geräts ein neues Update herausbringt und dieses verteilen will. Der Händler sendet das Update an den Server und dieser verteilt es dann an seine Geräte, die das Update benötigen. Zur Vereinfachung erwartet das Gerät in der Konzeption und der PoC-Implementierung dieser Arbeit nur Updates von einem Server und nicht von weiteren Servern.

3.1.3. Klassen des Geräts und Servers

Zur Prozessmodellierung wird der UML-Diagrammtyp *Sequenzdiagramm* gewählt. Um Objekte für die Sequenzdiagramme des Watchdog-Timer-Protokolls zu erhalten, werden zunächst Klassen gesammelt, die aus dem Ablaufdiagramm in Abbildung 3.1 hervorgehen. Jede Klasse hat eine eigene Aufgabe.

Es gibt zwei Hauptklassen: *Device* und *Server*. Die Klasse *Device* hat folgende Unterklassen mit der jeweils aufgeführten Funktion:

- *Boot*: Das Gerät wird neu gestartet und führt den Bootprozess durch.
- *AWDT_Init*: Initialisiert den Watchdog-Timer, damit dieser anfängt einen Zähler herunterzuzählen.
- *AWDT_GetNonce*: Generiert eine Nonce für eine Deferralticket-Anfrage und sendet diese Anfrage an den *Signer*.
- *AWDT_PutTicket*: Erhält ein Deferralticket vom *Server* und erhöht den Zähler, um das Timeout des Watchdog-Timer hinauszuzögern.
- *Timer*: Der *Timer* zählt herunter und überprüft in einem festgelegten Zeitintervall, ob die Zeit abgelaufen ist.
- *Sensor*: Misst bestimmte Werte in einem festgelegten Zeitintervall - z.B. Temperaturen - und sendet diese an den *Signer*.
- *StagingArea*: Dort werden Updates und Boottickets gespeichert. Von dort wird beim Empfang dieser Nachrichten ein Reset getriggert oder nach einem Neustart der *Verifier* aufgerufen, um die Validität der Updates und Boottickets zu überprüfen.
- *UpdateDownloader*: Generiert Anfragen für Boottickets oder Updates und sendet diese an den *Signer*.
- *Shutdown*: Resettet das Gerät.

Die Klasse *Server* hat folgende Unterklassen mit der jeweils aufgeführten Funktion:

- *Storage*: Speichert die Werte, die der *Sensor* des Geräts gemessen hat.
- *BootticketGenerator*: Generiert ein Bootticket für das Gerät und sendet es an den *Signer*.
- *DeferralticketGenerator*: Generiert ein Deferralticket für das Gerät und sendet es an den *Signer*.
- *UpdateGenerator*: Generiert ein Update für das Gerät und sendet es an den *Signer*.

Beide Klassen *Device* und *Server* haben jeweils folgende Unterklassen mit den aufgeführten Funktionen:

- *Signer*: Signiert jede Nachricht, die er erhält und je nach Nachrichtentyp, wird die signierte Nachricht an den entsprechenden Aktor gesendet.
- *Verifier*: Verifiziert jede Nachricht, die er erhält und je nach Nachrichtentyp, wird die verifizierte Nachricht an den entsprechenden Aktor gesendet.

3.1.4. Szenarien

Mithilfe des Protokollablaufs (Abbildung 3.1) und den aufgeführten Klassen werden die einzelnen möglichen Szenarien herausgearbeitet, die das Watchdog-Timer-Protokoll abhandeln soll. Die Abläufe, die in Abbildung 3.1 in Abhängigkeit zueinander aufgezeichnet sind, werden in kleinere Szenarien aufgeteilt sowie grafisch und zeitlich in Sequenzdiagrammen dargestellt. Die Sequenzdiagramme dienen der übersichtlichen Darstellung der zeitlichen Abläufe der Szenarien und des Protokolls. Es wurde nicht darauf geachtet, dass die Vorgaben zur Erstellung nach Unified Modeling Language (UML) umgesetzt wurden.

Im Folgenden werden nur die Szenarien betrachtet, in denen kryptografische Primitive verwendet werden, um die Hauptfunktionalität des Protokolls - die Funktion des Watchdog-Timers - umzusetzen. Dafür werden Boottickets, Updates und Deferraltickets benötigt. In Abschnitt 3.1.5 werden die verschiedenen Nachrichtentypen näher erläutert. Die blauen Kästchen stellen Objektklassen der Klasse `Device` dar und die grauen Kästchen stellen Objektklassen der Klasse `Server` dar.

Die Schlüsselpaare von Gerät und Server, die für die Erstellung und Verifizierung von Signaturen benötigt werden, wurden bereits erstellt und im sicheren Speicher des jeweils anderen Kommunikationspartner gespeichert.

3.1.4.1. Grundlegendes Szenario: Booten des Geräts bis zur Staging Area

Das Gerät startet sich selbst neu. Nach dem Booten wird die Staging Area auf ihre Inhalte überprüft. Dieser Ablauf (Abbildung 3.2) ist der Beginn der Szenarien 1, 2, 3, 6 und 7.

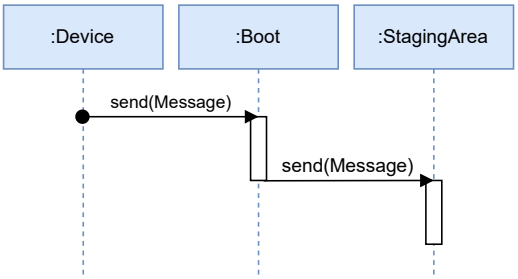

Abbildung 3.2.: Grundlegendes Szenario: Booten des Geräts.

3.1.4.2. Szenario 1: Kein Bootticket vorhanden

Nachdem das grundlegende Szenario (Abbildung 3.2) erfolgt ist, überprüft das Gerät, ob in der Staging Area ein Update vorhanden ist. Falls nicht, wird als nächstes überprüft, ob ein Bootticket in der Staging Area vorhanden ist.

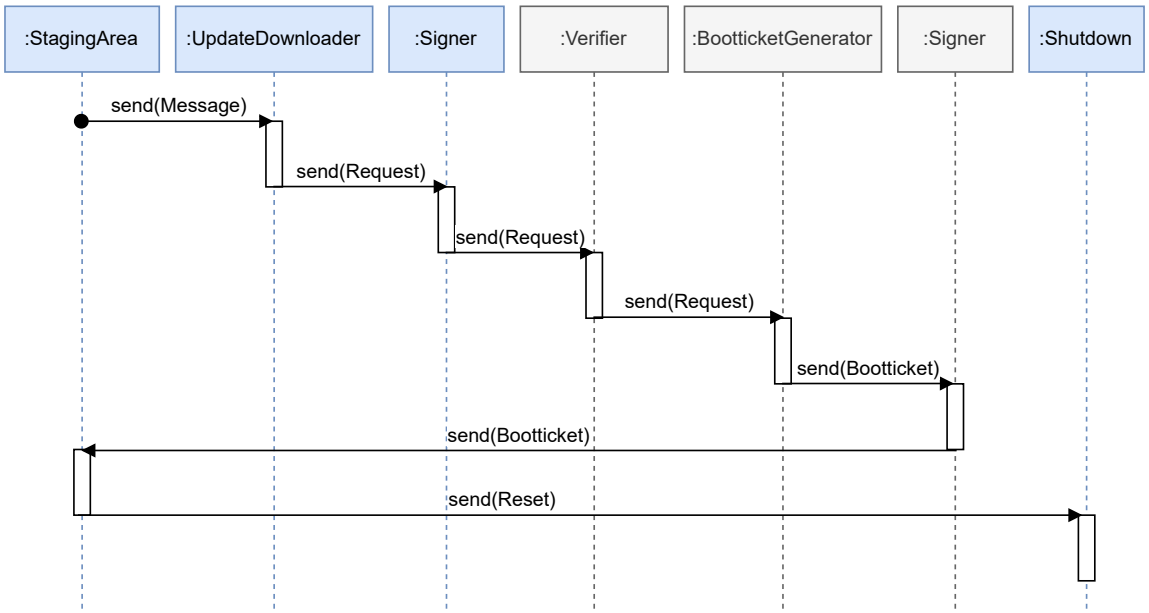

Abbildung 3.3.: Szenario 1: Kein Bootticket in der Staging Area vorhanden. Blaue Kästchen: Device-Klassen. Graue Kästchen: Server-Klassen.

Die Business-Logik kann nur mithilfe eines validen Boottickets ausgeführt werden. Abbildung 3.3 zeigt, dass wenn kein Bootticket vorhanden ist, vom sogenannten Update-Downloader des Geräts eine signierte Bootticket-Anfrage (Request) an den Server gesendet

wird. Dazu wird außerdem eine Nonce mit einem Hardware Random Number Generator (HRNG) generiert und mitsigniert. Die Nonce wird zusätzlich im sicheren Gerätespeicher gespeichert. Der Server verifiziert die Anfrage, erstellt ein Bootticket, signiert es und sendet es an das Gerät. Das Gerät speichert das Bootticket in der Staging Area und führt anschließend einen Reset durch.

3.1.4.3. Szenario 2: Bootticket vorhanden, aber nicht valide

Nachdem das grundlegende Szenario (Abbildung 3.2) erfolgt ist, überprüft das Gerät, ob in der Staging Area ein Update vorhanden ist. In Szenario 2 (Abbildung 3.4) liegt kein Update vor, daher wird die Staging Area auf das Vorhandensein eines Boottickets überprüft, das mithilfe des öffentlichen Schlüssels des Servers, von dem das Bootticket erwartet wird, verifiziert wird.

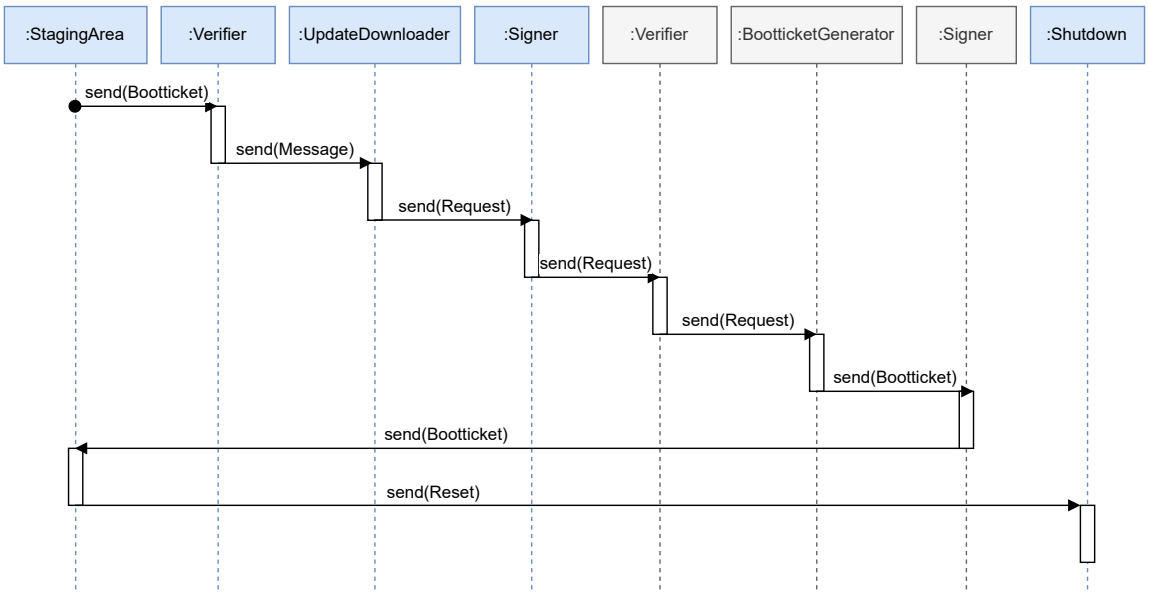

Abbildung 3.4.: Szenario 2: Bootticket in der Staging Area vorhanden, aber nicht valide. Blaue Kästchen: Device-Klassen. Graue Kästchen: Server-Klassen.

Die Gründe dafür, dass das Bootticket nicht verifiziert werden kann, sind entweder:

- Die Signatur kann nicht verifiziert werden, da der öffentliche Schlüssel nicht korrekt ist (Authentizität nicht gegeben).
- Die Integrität der Nachricht ist nicht gegeben, da der selbst generierte Hash des Boottickets nicht mit dem Hash des empfangenen Boottickets übereinstimmt und somit die Signatur nicht verifiziert werden kann.
- Die in Szenario 1 generierte Nonce ist nicht identisch mit der Nonce des empfangenen Boottickets (Integrität nicht gegeben).

Da das Bootticket nicht valide ist, fragt das Gerät ein neues Bootticket beim Server an. Dies geschieht wie in Szenario 1 beschrieben vom Update-Downloader an.

3.1.4.4. Szenario 3: Bootticket vorhanden und valide

Zunächst erfolgt das grundlegende Szenario (Abbildung 3.2). Da in diesem Szenario kein Update vorhanden ist, wird in den Abbildungen 3.5 und 3.6 die Staging Area auf das Vorhandensein eines Boottickets überprüft, das mithilfe des öffentlichen Schlüssels des Servers, von dem das Bootticket erwartet wird, verifiziert wird. Nachdem die Signatur verifiziert wurde, wird die in Szenario 1 generierte Nonce verifiziert, indem sie mit der in Szenario 1 gespeicherten Nonce verglichen wird. Sind die gespeicherte und die empfangene Nonce gleich, so ist das gesamte Bootticket verifiziert und gilt als valide. Anschließend startet das Gerät zeitgleich den AWDT (Abbildung 3.5) mit *AWDT_Init* und die Business-Logik (Abbildung 3.6).

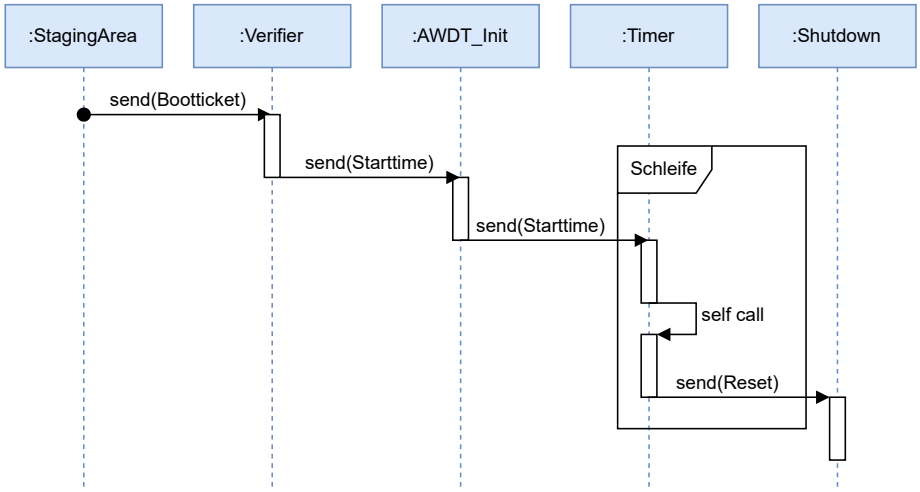

Abbildung 3.5.: Szenario 3a: Bootticket in der Staging Area vorhanden und valide: Mit dem Start in die Business-Logik wird der Timer des Geräts gestartet.

In Abbildung 3.5 wird der Timer gestartet. In festgelegten Zeitintervallen überprüft er sich selbst, ob die Zeit abgelaufen ist. Falls die Zeit abgelaufen ist, wird das Gerät heruntergefahren und neu gestartet.

Als Business-Logik wird in dieser Arbeit ein *Sensor* gewählt, der zufällig eine Zahl zwischen 4 und 14 wählt und als Messwert festlegt. In Abbildung 3.6 wird dieser Sensor gestartet, der sich selbst in einem festgelegten Zeitintervall immer wieder auffordert eine Messung aufzunehmen, diese mit dem Zeitstempel der Messung zu signieren und an den Server zu senden. Der Server verifiziert die Nachricht mit dem Sensor-Messwert. Anschließend speichert der Server den Messwert im *Storage* oder verarbeitet ihn weiter.

In Szenario 3 wird kein Deferralticket gesendet, das heißt, wenn der Timer abgelaufen ist, erfolgt ein Shutdown des Geräts (vgl. Abbildung 3.5). Damit der Timer nicht abläuft, werden die Szenarien 4 und 5 ergänzt.

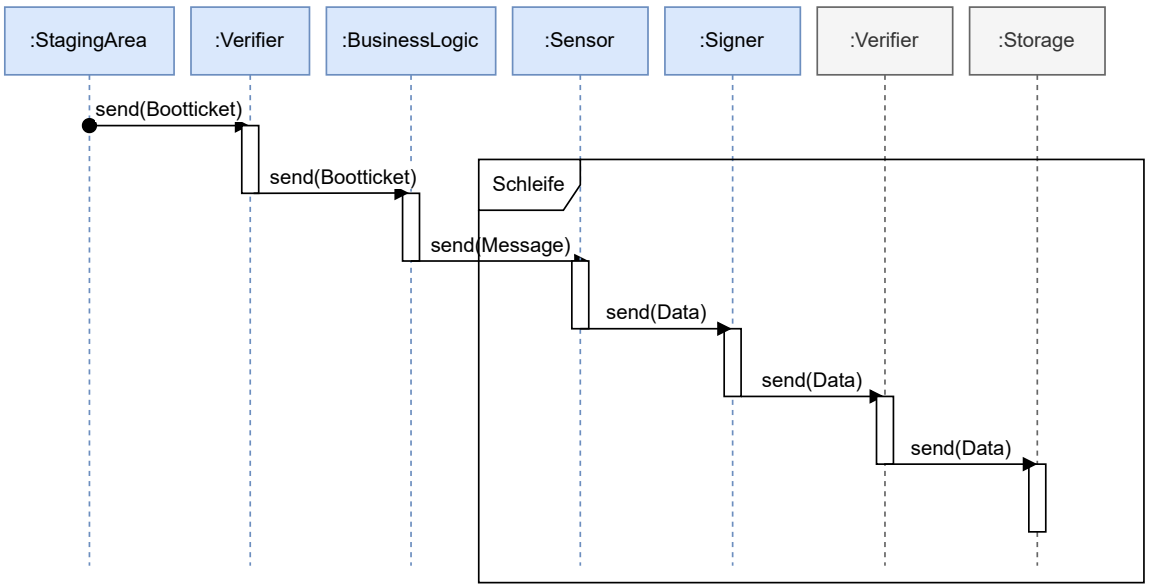

Abbildung 3.6.: Szenario 3b: Bootticket in der Staging Area vorhanden und valide: Mit dem Start in die Business-Logik wird der Sensor gestartet. Blaue Kästchen: Device-Klassen. Graue Kästchen: Server-Klassen.

3.1.4.5. Szenario 4: Business-Logik mit Deferralticket

Der Ablauf in Abbildung 3.7 geht aus Szenario 3 hervor. Der Timer und die Business-Logik mit dem Sensor werden gestartet und laufen parallel zu Szenario 4 ab.

In Abbildung 3.7 wird die Business-Logik nach einem festgelegten Zeitintervall getriggert und erstellt mit *AWDT_GetNonce* eine Deferralticket-Anfrage (Request) mit einer neu generierten Nonce. Die Nachricht wird im Anschluss signiert und an den Server gesendet. Wurde die Nachricht erfolgreich vom Server verifiziert, wird ein Deferralticket generiert und signiert an das Gerät gesendet. Das Gerät verifiziert das Deferralticket und wendet es mit *AWDT_PutTicket* an. Der Timer wird erhöht und überprüft in festgelegten Zeitintervallen, ob der Timer abgelaufen ist. Falls der Timer abgelaufen ist, wird das Gerät heruntergefahren. Der Fall des Timeouts des Watchdog-Timers tritt in diesem Szenario nicht auf, da in festgelegten Abständen immer wieder neue Deferraltickets angefragt und vom Server an das Gerät gesendet werden. Das gewünschte Deferralticket wird nicht gesendet, wenn der Server die Verlängerung aufgrund von z.B. einer Kompromittierung des Geräts verweigert. Sollte es zu einem Timeout kommen, wird das Gerät geresettet.

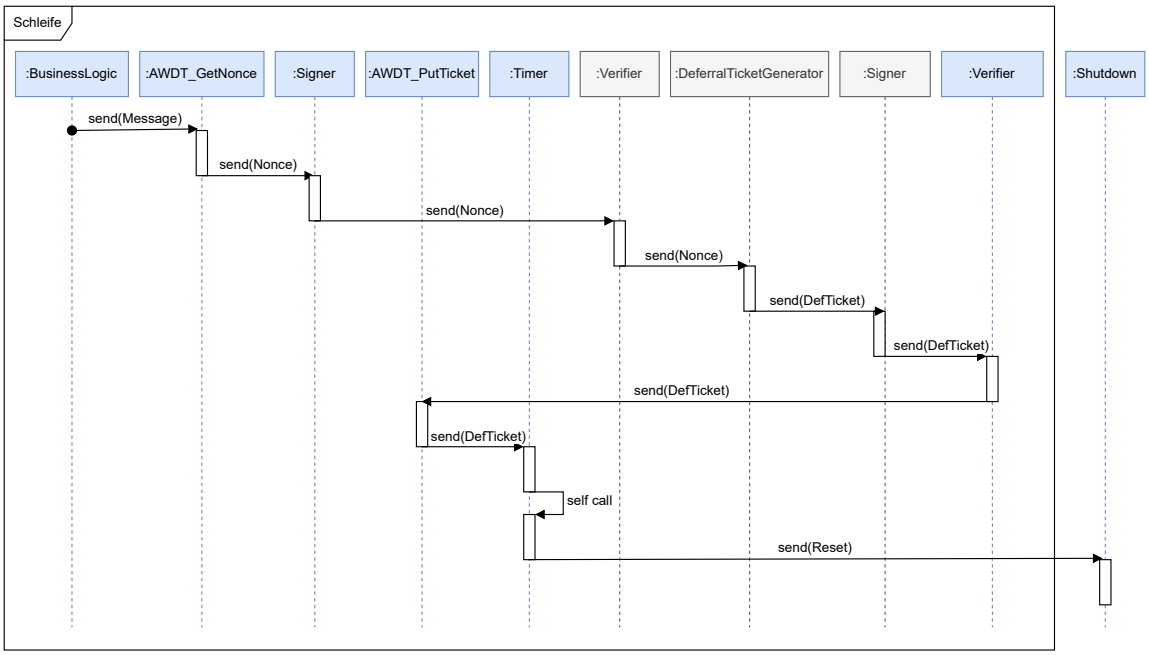

Abbildung 3.7.: Szenario 4: Business-Logik mit Deferralticket. Blaue Kästchen: Device-Klassen. Graue Kästchen: Server-Klassen.

3.1.4.6. Szenario 5: Business-Logik mit invalidem Deferralticket

Szenario 5 geht aus Szenario 3 hervor. Abbildung 3.8 beginnt mit der Business-Logik neben der parallel der Timer und der Sensor gestartet werden.

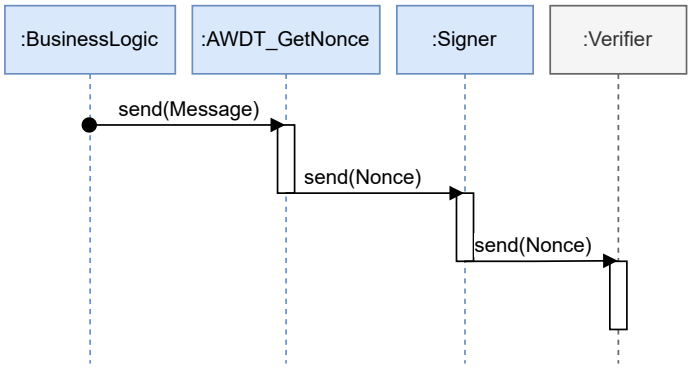

Abbildung 3.8.: Szenario 5: Business-Logik mit invalidem Deferralticket. Blaue Kästchen: Device-Klassen. Graue Kästchen: Server-Klassen.

In diesem Fall erstellt die Business-Logik wie in Szenario 4 eine Deferralticket-Anfrage (Request) mit Nonce und signiert diese. Nachdem die signierte Nachricht an den Server gesendet wurde, versucht der Server die Nachricht zu verifizieren. Die Verifizierung ist nicht

möglich, weil dem Server bekannt ist, dass das Gerät kompromittiert wurde oder die Signatur der Nachricht nicht stimmt. In diesem Fall generiert der Server kein Deferralticket und beendet somit das Szenario.
Da das Gerät ohne Deferralticket das Timeout nicht hinauszögern kann, wird das Gerät geresettet. Der Resetvorgang wird in Abbildung 3.8 nicht dargestellt, kann aber aus Abbildung 3.7 ab dem Timer eingesehen werden.

3.1.4.7. Szenario 6: Update vorhanden, aber nicht valide

Nachdem das grundlegende Szenario (Abbildung 3.2) erfolgt ist, überprüft das Gerät, ob in der Staging Area ein Update vorhanden ist. In diesem Szenario ist ein Update vorhanden. Aus Abbildung 3.9 geht hervor, dass das Update zur Überprüfung an den Verifizierer gesendet wird. Der Verifizierer überprüft die Versionsnummer und die Signatur des Updates.

Das Update kann nicht verifiziert werden. Die Gründe dafür sind z.B.:

- Die empfangene und gespeicherte Versionssnummer können nicht miteinander verifiziert werden.
- Die Signatur kann nicht verifiziert werden, da der öffentliche Schlüssel nicht korrekt ist (Authentizität nicht gegeben).
- Die Integrität der Nachricht ist nicht gegeben, da der selbst generierte Hash des Updates nicht mit dem Hash des empfangenen Updates übereinstimmt und somit die Signatur nicht verifiziert werden kann.

Wenn der Vergleich der Versionsnummern oder die Verifizierung der Signatur nicht möglich ist, gilt das Update als invalide.

Der Update-Downloader wird beauftragt eine Update-Anfrage (Request) zu erstellen. Die Anfrage wird signiert und an den Server gesendet. Der Server verifiziert die Anfrage des Geräts mit dem öffentlichen Schlüssel des Geräts. Falls ein Update bereitsteht, generiert der Server ein neues Update. Falls kein neues Update bereitsteht, kann der Server zum Beispiel die aktuelle Version erneut senden. Das Update wird vom Server in der Staging Area des Geräts gespeichert. Das Speichern eines Elements in der Staging Area löst einen Reset des Geräts aus.
Der Fall, dass kein Update bereitsteht, wird in dieser Konzeption nicht berücksichtigt.

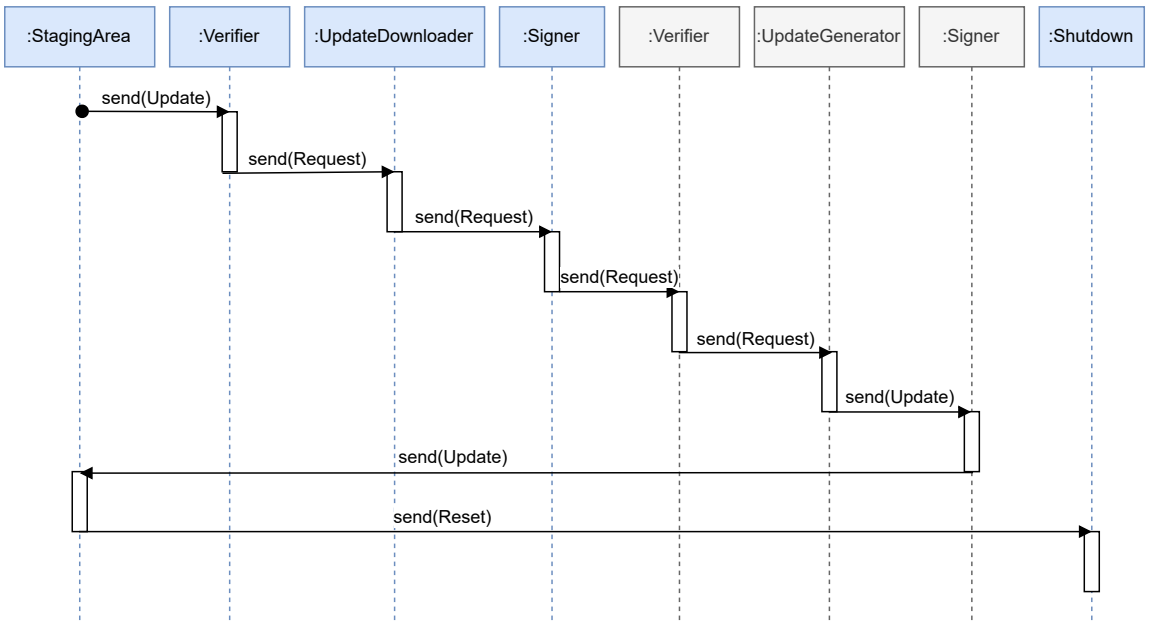

Abbildung 3.9.: Szenario 6: Update in Staging Area vorhanden, aber nicht valide. Blaue Kästchen: Device-Klassen. Graue Kästchen: Server-Klassen.

3.1.4.8. Szenario 7: Update vorhanden und valide

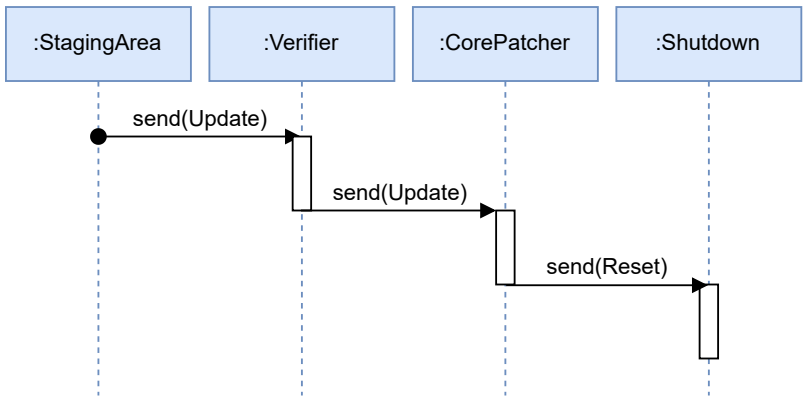

Abbildung 3.10.: Szenario 7: Update in Staging Area des Geräts vorhanden und valide.

Nachdem das grundlegende Szenario (Abbildung 3.2) erfolgt ist, überprüft das Gerät, ob in der Staging Area ein Update vorhanden ist. Aus Abbildung 3.10 geht hervor, dass ein Update vorliegt und dieses zur Überprüfung an den Verifizierer gesendet wird. Der Verifizierer überprüft die Versionsnummer und die Signatur des Updates. In diesem Szenario konnte das Update verifiziert werden und gilt somit als valide. Der Core-Patcher installiert das Update und das Gerät wird neu gestartet.

3.1.4.9. Szenario 8: Server sendet Update

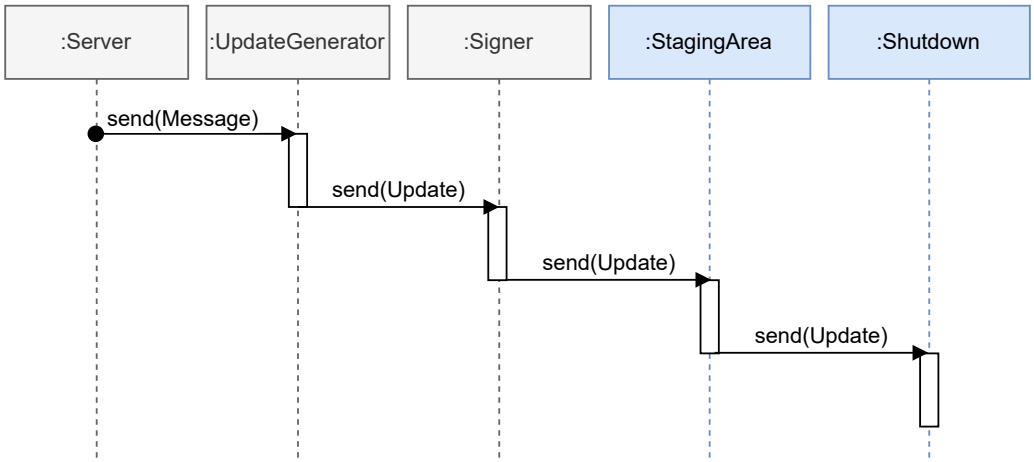

Abbildung 3.11.: Szenario 8: Server sendet Update an das Gerät. Blaue Kästchen: Device-Klassen. Graue Kästchen: Server-Klassen.

Abbildung 3.11 zeigt, dass der Server mithilfe des Update-Generators ein Update erstellt hat. Der Server signiert das Update und sendet es an die Staging Area des Geräts. Das Gerät speichert das Update im Speicher der Staging Area und startet sich neu.

3.1.5. Nachrichtentypen und ihre Schutzziele

In den Watchdog-Timer-Protokollen von Huber *et al.* [2] und Xu *et al.* [3] werden folgende Nachrichten versendet:

- Verschiedene *Updates* vom Server an das Gerät
- *Boottickets* vom Server an das Gerät
- *Deferraltickets* vom Server an das Gerät
- Monitoring-Daten oder andere *Messdaten* vom Gerät an den Server
- *Noncen* werden vom Gerät an den Server gesendet und wieder zurück um Integrität der Daten zu gewährleisten

Diese Nachrichtentypen werden auch in das neue Konzept des Watchdog-Timer-Protokolls übernommen. Es werden noch weitere Nachrichtentypen wie *Request*, *Addresses* und *Message* ergänzt, wobei letztere in Abschnitt 4.4 beschrieben werden. Ein *Request* ist eine Anfrage, die vom Gerät an den Server gesendet wird. So kann das Gerät entweder ein Bootticket, ein Deferralticket oder ein Update beim Server anfragen. Für die Anfrage eines Boot- oder Deferraltickets wird eine vom Gerät generierte Nonce mitgesendet, damit das Gerät später verifizieren kann, dass die Nachricht von dem Server gesendet wurde, an

den auch die Anfrage gesendet wurde. Noncen dienen außerdem dem Schutz vor Replay-Angriffen (Abschnitte 2.3.4). In Abschnitt 4.4 wird beschrieben wie die Nachrichtentypen in der Implementierung umgesetzt werden.

Die zu versendenden Nachrichtentypen sollen durch kryptografische Primitive geschützt werden. Im Paper von Xu *et al.* [3] wird *Verfügbarkeit* als eines der zentralen Schutzziele dargestellt. Des Weiteren werden *Vertraulichkeit* und *Integrität* des Systems *CIDER*, sowie die Authentifizierung und Attestierung der Kommunikationspartner genannt [3]. Auch Huber *et al.* [2] schreiben von Integrität, Verfügbarkeit, Attestierung und Authentifizierung. Die einzelnen Schutzziele wurden in Abschnitt 2.1.1 erläutert. Für die Herstellung der *Authentizität* werden Verfahren der Authentifizierung und Attestierung verwendet. Wie bereits in der Abgrenzung in Abschnitt 3.1.1 erklärt, wird die Attestierung in dieser Arbeit nicht betrachtet. Die Verfügbarkeit soll für das gesamte System und Vertraulichkeit, Integrität und Authentizität für jede Nachricht gewährleistet werden. Die Umsetzung dieser Schutzziele wird im nachfolgenden Abschnitt 3.1.6 beschrieben.

3.1.6. Verwendete kryptografische Primitive und Algorithmen

In Tabelle 3.1 werden die technischen und organisatorischen Maßnahmen zusammengetragen, die bei CIDER [3] und Lazarus [2] angewendet werden, um die in Abschnitt 3.1.5 aufgeführten Schutzzziele zu erreichen.

Schutzziel	Technische Maßnahmen	Organisatorische Maßnahmen
Vertraulichkeit	Verschlüsselung der Kommunikation über SSL, Festplattenverschlüsselung	Sichere Speicherbereiche, Geheimnisse werden nicht versendet und es wird nur von sicherer Software darauf zugegriffen
Integrität	Hashen der Geräte-Firmware, Write-Latch	Versionsnummer oder Nonce für Freshness mitsenden
Verfügbarkeit	Schnelle Wiederherstellung durch Reset	Watchdog-Timer: Reset, wenn kein Deferralticket und kein Bootticket vorhanden ist
Authentizität (Identität)	Digitale Signatur (ECDSA)	-

Tabelle 3.1.: Übersicht über die technischen und organisatorischen Maßnahmen, die bei den Protokollen CIDER [3] und Lazarus [2] verwendet werden - ohne Berücksichtigung der Primitive zum Umsetzen von DICE.

Vertraulichkeit kann durch symmetrische Verschlüsselung erreicht werden. Laut Xu *et al.* [3] reichen Kryptografie und Protokollcode wie z.B. SSL oder verschlüsselte Festplatten aus, um die Vertraulichkeit und Integrität der Daten, „die an nicht-vertrauenswürdige Speicher- und Netzwerkgeräte gesendet werden," zu schützen. Im Paper von Huber *et al.* [2] wird

beschrieben, dass manche Schlüssel vor der Übertragung an den Server verschlüsselt werden - eine Verschlüsselung der versendeten Nachrichten wird nicht erwähnt. Daher wird bei der Durchführung dieser Arbeit davon ausgegangen, dass die Nachrichten nicht verschlüsselt werden müssen, da die Vertraulichkeit kein Kriterium beim Lazarus-Protokoll ist. Es wird jedoch davon ausgegangen, dass die verwendete SSL-Verbindung sicher ist und die Festplatten verschlüsselt werden.

Die *Integrität* und die *Authentizität von Identitäten* der Kommunikationspartner wird bei CIDER [3] und Lazarus [2] mithilfe digitaler Signaturverfahren sichergestellt (vgl. Abschnitt 2.3.1.3). Bevor Updates installiert werden, wird die Herkunft des Updates überprüft, indem der Sender mithilfe der Signatur der Nachricht verifiziert wird. Updates und Boottickets werden nur nach einem Neustart überprüft und durchgeführt da der Systemzustand nach einem sogenannten ordentlichen Neustart als sauber bezeichnet werden kann [2]. Boottickets und Deferraltickets können nur mithilfe von Noncen und Signaturen verifiziert werden. Xu *et al.* [3] nutzen das digitale Signaturverfahren *Ed25519* zum Sicherstellen der Authentizität der Identität der Kommunikationspartner (vgl. Tabelle 3.1). Des Weiteren werden SHA2-256-Komponenten der formal-verifizierten High-Assurance Cryptographic Library (HACL) verwendet [3].

Der Server bei CIDER entscheidet aufgrund des Hashes der Geräte-Firmware und einer Gerät-spezifischen Identifikationsnummer, ob er ein Deferralticket für dieses Gerät erstellt, dies wird u.a. mit DICE umgesetzt. Der Hash der Geräte-Firmware wurde vom Server vor dem Verteilen der Firmware auch erstellt und gespeichert. [3]

Wie in Tabelle 3.1 zusammengetragen, wurde bei der Implementierung des Prototyps von Lazarus [35] *ECDSA-secp256r1 Sign + Verify* [2] verwendet, um die Integrität und Authentizität der Nachrichten des Geräts sicherzustellen. Zum Signieren und Verifizieren wurde beim Lazarus-Prototyp [35] mit dem Modul `pk` gearbeitet, das ECDSA zur Authentifizierung der Kommunikationspartner nutzt (siehe Abschnitt 2.3.1.4). Die zum Signieren verwendete Funktion heißt `mbedtls_pk_sign` und die zum Verifizieren `mbedtls_pk_verify`. Diese Funktionen verwendeten digitale Signaturverfahren mit elliptischen Kurven. Huber *et al.* [2] schreiben davon, dass Lazarus die elliptische Kurve *prime256v1* (auch bekannt als *secp256r1*) zum Herleiten von Geräteschlüsselpaaren einsetzt. Die öffentlichen Schlüssel dieser Kurve sind in einem Zertifikat des Typs X.509 inbegriffen. Des Weiteren verwendet Lazarus *HMAC-SHA256* zum Herleiten weiterer interner Geheimnisse beim Boot, die für DICE und DICE++ relevant sind.

Die *Verfügbarkeit* soll durch eine schnelle Wiederherstellung der Funktionsfähigkeit des IoT-Geräts durch den Reset erfolgen, der durch den Ablauf eines Watchdog-Timers bzq. Authenticated Watchdog Timers (AWDTs) getriggert wird. Hier spielt der Watchdog-Timer bzw. Authenticated Watchdog Timer der eine große Rolle, da durch ihn ein Neustart des IoT-Geräts hervorgerufen werden kann und das Gerät somit schnell wieder verfügbar ist. Verfügbarkeit wird in CIDER durch die zusätzliche Verwendung von Speichertreiber, Netzwerk-Stack und geräteabhängigem Low-Level Initialisierungscode erreicht [3]. Damit die Firmware selbst bis zum nächsten Reset keine Schutzmaßnahmen außer Kraft setzen kann - falls sie von einem Angreifer kontrolliert wird - werden Hardware-Latches verwendet. Zwei weitere Primitive, die CIDER verwendet sind *Gated Boot* und *Reset Trigger* [3].

Gated Boot garantiert, dass das Gerät nur in die Firmware bootet, die zu diesem Zeitpunkt vom Server authorisiert wurde. Wenn ein Angreifer die Kontrolle übernommen hat, muss ein Reset Trigger implementiert sein, der das Gerät neu startet. Der Reset Trigger ist bei CIDER ein AWDT.

Beim CIDER-Protokoll [3] wurden weitere Schutzmaßnahmen angewandt, um die Schutzziele zu gewährleisten. Der Fokus von CIDER liegt auf der Dominanz, die laut Xu *et al.* [3] als eine stärkere Version von Attestierung angesehen werden kann, da sie dem Backend die Möglichkeit gibt die Software eines Geräts zu steuern, anstatt sie nur zu identifizieren - wie es bei der Attestierung der Fall ist. Für die Attestierung und Authentifizierung des Geräts wird DICE verwendet. Bei Lazarus wird ein HRNG zum Erstellen von Noncen eingesetzt, um vor verfälschten Nachrichten zu schützen und Freshness zu gewährleisten [2]. CIDER verwendet Noncen, um sich vor Replay-Angriffen zu schützen [3].

3.1.7. Das abgewandelte Watchdog-Timer-Protokoll

In diesem Abschnitt werden die kryptografischen Primitiven von CIDER und Lazarus aufgeführt, die in das abgewandelte zunächst klassische Watchdog-Timer-Protokoll übernommen werden. Des Weiteren wird begründet, welche Konzepte nicht übernommen werden.

Bei der *Vertraulichkeit* werden die technischen und organisatorischen Maßnahmen aus CIDER und Lazarus als ausreichend angenommen (vgl. Tabelle 3.1). Es wird bei der Konzeption und Implementierung des abgewandelten Watchdog-Timer-Protokolls davon ausgegangen, dass eine Festplattenverschlüsselung vorliegt und die Verschlüsselung der Kommunikation übers Internet sicher ist.

Die *Integrität* wird im abgewandelten Watchdog-Timer-Protokoll durch die Verwendung einer Hashfunktion im Signaturalgorithmus sichergestellt. Die verwendete Hashfunktion ist abhängig vom Algorithmus. Des Weiteren wird ein HRNG verwendet, um Noncen für Boottickets und Deferraltickets zu erstellen und somit vor Replay-Angriffen und gefälschten Nachrichten zu schützen. Bei Updates werden die Versionsnummern mitgesendet und überprüft.

Um zu überprüfen, ob die *Authentizität der Daten und Identitäten* der Kommunikationspartner gewährleistet ist, wird die Nachricht - bzw. der daraus generierte Hash - mithilfe des Signatualgorithmus und privaten Schlüssel des Senders signiert und kann vom Empfänger mit dem öffentlichen Schlüssel des Senders die Signatur verifizieren.

Die Signatur wird bei neu konzipierten Watchdog-Timer-Protokoll mithilfe des digitalen Signaturalgorithmus Elliptic Curve Digital Signature Algorithm (ECDSA), der auf einer elliptischen Kurve basiert, erstellt. Speziell werden die Kurve *secp256r1* sowie die Verfahren *RSA-2048* und *RSA-4096* testweise bei der Proof-of-Concept-Implementierung eingesetzt, bei denen die zu verwendende Hashfunktion als Parameter mitangegeben werden kann. HMAC-SHA256 wurde bei Lazarus verwendet, um DICE- und DICE++-spezifische Schlüssel herzuleiten [2]. Da weder DICE- noch DICE++ im abgewandelten Watchdog-Timer-Protokoll umgesetzt werden, wird HMAC-SHA256 nicht implementiert.

Die *Verfügbarkeit* des Systems wird mithilfe des Watchdog-Timers erreicht, der das IoT-Gerät neu startet, wenn der Timer abgelaufen ist. Der Timer kann Deferraltickets vom Server erhalten und so das Timeout und die Zeit des Resets auf einen späteren Zeitpunkt verschieben. Sollte das Gerät jedoch kompromittiert sein, wird somit garantiert, dass ein Timeout schneller eintritt. Mit einem Neustart wird das Gerät in einen sicheren Zustand versetzt, die Verbindung zum Server kann wieder hergestellt werden und die Nachrichtenversendung ist wieder möglich.

Bei der Konzeption des abgewandelten Watchdog-Timer-Protokolls werden keine Algorithmen zur Attestierung betrachtet (vgl. Abschnitt 3.1.1). Des Weiteren setzten Huber *et al.* [2] das Lazarus-System mithilfe einer Trusted Execution Environment (TEE) um, damit alle sicherheitskritischen Funktionen innerhalb dieser Umgebung - und somit isoliert von nichtvertrauenswürdiger Software - durchgeführt werden können. Die Download-Funktionalität für Boottickets und Updates vom Server sowie die Business-Logik laufen außerhalb der TEE. In dieser Arbeit wird davon ausgegangen, dass der später zu verwendende Mikroprozessor eine TEE besitzt und sicher ist, obwohl Lapid und Wool [68] gezeigt haben, dass Seitenkanal-Angriffe auf TEEs möglich sind. Es wird angenommen, dass die Isolierung prinzipiell möglich ist. Im abgewandelten Protokoll wird keine TEE verwendet, da dies außerhalb des Rahmens dieser Arbeit liegt.

In Bezug auf das abgewandelte Watchdog-Timer-Protokoll wird angenommen, dass die Umgebung, in der das Device Provisioning (die Erstbereitstellung des Geräts) durchgeführt wurde, sicher war und die öffentlichen Schlüssel sicher zwischen Gerät und Server ausgetauscht wurden (vgl. Abschnitt 3.1.4). Das Konzept und die Implementierung sieht demnach kein Device Provisioning vor und es wird vorab ein Schlüsselpaar für jeden Kommunikationspartner erstellt und im jeweiligen Gerätespeicher abgelegt.

In der realen Welt gibt es nicht nur einen Server, der Geräten Updates oder andere Nachrichten senden kann. Es könnte z.B. einen Server geben, von dem ausschließlich Updates gesendet werden und einen von dem ausschließlich Boottickets oder Deferraltickets gesendet werden. Damit jeder dieser Server den Geräten Nachrichten senden kann, muss zur Authentifizierung dieser weiteren Server deren jeweiliger öffentlicher Schlüssel auf den Geräten gespeichert sein. Das neu konzipierte Watchdog-Timer-Protokoll beschränkt sich jedoch auf die Kommunikation zwischen einem Server und einem Gerät.

Des Weiteren könnten mehrere voneinander unabhängige Geräte mit gleicher Funktionalität, parallel nebeneinander laufen und mit demselben Server kommunizieren. Die Einbindung weiterer möglicher Geräte und Server liegt außerhalb des Rahmens dieser Arbeit.

3.2. Quantensicheres Watchdog-Timer-Protokoll

In diesem Abschnitt werden zunächst die Angriffsflächen des neu konzipierten Watchdog-Timer-Protokolls beschrieben (Abschnitt 3.2.1). Um die Angriffsflächen des Protokolls zu verringern, wird anschließend analysiert, inwiefern die im klassischen Watchdog-Timer-Protokoll verwendeten kryptografischen Primitive (vgl. Abschnitt 3.1.6) ersetzt und somit quantensicher gemacht werden können (Abschnitt 3.2.2).

3.2.1. Angreifermodell und Systemannahmen

Es wird angenommen, dass der Angreifer keinen physischen Zugriff auf das IoT-Gerät oder den Server hat, womit Seitenkanal-Angriffe in dieser Arbeit nicht betrachtet werden. Des Weiteren wird davon ausgegangen, dass der Angreifer einen Quantencomputer zur Verfügung hat. Daher muss das Watchdog-Timer-Protokoll mit quantenresistenten Algorithmen implementiert werden, wobei die Annahme besteht, dass die quantensicheren Algorithmen nicht gebrochen werden können. Es wird davon ausgegangen, dass der verwendete Server genügend Speicherplatz hat, sowie vertrauenswürdig und sicher gegenüber Angriffen ist. Des Weiteren kann davon ausgegangen werden, dass die Angreifer sich einen Fernzugriff auf das IoT-Gerät verschafft haben und es manipulieren können. Dies kann durch Security Information and Event Management (SIEM) bzw. den Server festgestellt werden. Ziel ist in dieser Arbeit nicht die Vermeidung oder Erkennung solcher Angriffe, sondern die schnelle Wiederherstellung des IoT-Geräts mithilfe des Watchdog-Timer-Protokolls.

3.2.2. Ersetzung der klassischen kryptografischen Algorithmen durch PQC-Algorithmen

In Tabelle 3.1 werden die kryptografischen Primitiven aufgeführt, die bei Lazarus [2] und CIDER [3] verwendet wurden. Des Weiteren wurden in Abschnitt 3.1.7 die klassischen Primitiven beschrieben, die im neu konzipierten Watchdog-Timer-Protokoll verwendet werden. Von diesen Primitiven müssen einige ersetzt werden, da sie durch Quantencomputer gebrochen werden können.

Bei der in Lazarus [2] verwendeten elliptischen Kurve *prime256v1* (auch bekannt als: *secp256r1*) handelt es sich um ein digitales Signaturverfahren und somit um einen asymmetrischen Algorithmus, der gebrochen werden kann. In Tabelle 3.2 werden PQC-Algorithmen aufgeführt, die die elliptische Kurve ersetzen könnten. Die Empfehlungen des BSI [7] stimmen mit den Finalisten des Standardisierungsprozesses des NIST für digitale Signaturverfahren überein [69]. Das NIST [13] hat im Jahr 2022 bekannt gegeben, dass die digitalen Signaturverfahren Dilithium, Falcon und SPHINCS+ standardisiert werden sollen. Aus diesem Grund wird sich in der folgenden Analyse und Diskussion ausschließlich mit den in Abschnitt 2.4.2 vorgestellten und in Tabelle 3.2 aufgeführten Verfahren auseinandergesetzt.

Da alle drei Verfahren standardisiert werden sollen, wird davon ausgegangen, dass sie die Anforderungen, die in Abschnitt 2.4.2.1 aufgeführt wurden, erfüllen. Dilithium (Abschnitt 2.4.2.3) wurde gewählt, da es eine hohe Sicherheit bietet und eine hervorragende Leistung erzielt. Das NIST geht davon aus, dass Dilithium in einer Mehrheit an Anwendungen gut funktionieren wird. Der Nachteil bei den Signaturen, die Dilithium erstellt, ist die Signaturgröße. Falcon (Abschnitt 2.4.2.4) wurde als Alternative ausgewählt, um einen Standard zu bieten, der für Anwendungsfälle geeignet ist, bei dem die Dilithium-Signaturen zu groß sind. Um neben gitterbasierten Verfahren auch eine hashbasierte Alternative zu haben, falls gitterbasierte Verfahren gebrochen werden, wurde SPHINCS+ (Abschnitt 2.4.2.5) ausgewählt. [13] Es wurde ein weiterer Aufruf für digitale Signaturverfahren gestartet, um weitere Alternativen zu finden.

Verfahren	Dilithium			Falcon		SPHINCS+		
Art	gitterbasiert			gitterbasiert		zustandslos, hashbasiert		
Hashfunktion	SHAKE-128, SHAKE-256			SHAKE-256		SHAKE-256, SHA-256		
Bitsicherheit	-	192	256	128	256	128	192	256
NIST-Sicherheitslevel	2	3	5	1	5	1	3	5
Signaturgröße (Bytes)	2.420	3.293	4.595	690 (666)	1.330 (1.280)	siehe Tabelle 2.3		
Public Key Size (Bytes)	1.312	1.952	2.592	897	1.793	32	48	64
Private Key Size (Bytes)	2.528	4.000	4.864	1.281	2.305	64	96	128

Tabelle 3.2.: Übersicht über die Eigenschaften der verschiedenen Parametersets von Dilithium [70], Falcon [71] und SPHINCS+ [67]. Die Signaturgrößen der unterschiedlichen SPHINCS+-Versionen sind in Tabelle 2.3 aufgeführt. Die Größen für die privaten Schlüssel für Dilithium und Falcon wurden mithilfe von `signer.details` der Bibliothek `liboqs-python` bestimmt. Die Signaturgrößen von Falcon, die bei `liboqs-python` verwendet werden, weichen von denen in der Spezifikation [71] ab, dies stehen in Klammern. Im Folgenden wird sich auf die Größen aus der Bibliothek bezogen.

Die Hashfunktionen, die in Tabelle 3.2 für jeden Algorithmus aufgeführt sind, sind Teil des Algorithmus und werden innerhalb der jeweiligen `sign-` und `verify`-Funktionen durchgeführt. Hashfunktionen können nicht von Quantencomputern gebrochen werden, wenn die resultierenden Hashes groß genug sind, um dem Grover Algorithmus Stand zu halten. Mit Grover's Algorithmus [7] wird die Berechnungszeit von Schlüsseln halbiert. Bei Hashfunktionen wird hingegen die Zeit reduziert, die ein Quantencomputer benötigt, um einen Kollisionsangriff erfolgreich durchzuführen. Bei SHA256 verringert sich die Sicherheit von 128 auf 80 Bit und bei SHA384 von 192 auf 128 Bit [72]. Laut BSI gilt eine Größenordnung von 2^{128} Quantenoperationen allerdings als nicht realisierbar, da hohe Anforderungen an die Quantencomputerschaltkreise gestellt werden, die die Implementierung einer Suche auf der Basis des Grover Algorithmus erschweren [7]. Des Weiteren wird bei SHA-256 davon ausgegangen, dass 13 Millionen Qubits benötigt werden, um eine Hashkollision innerhalb

eines Tages zu finden [73]. Aktuell arbeitet ein Quantenprozessor vom IBM mit 433 Qubits [61]. Preston [74] kam jedoch zu dem Ergebnis, dass SHA2-512 die kryptografische Hashfunktion ist, die am quantensichersten ist. Daher wird für die Messungen der Funktionen und Szenarien unter Verwendung klassischer Algorithmen die Hashfunktion SHA2-512 verwendet. Für die genannten quantensicheren Signatualgorithmen sind die Hashfunktionen bereits inkludiert.

4. Implementierung

In diesem Kapitel werden zunächst die Anforderungen an das System (Abschnitt 4.1) und die verwendeten Frameworks und Bibliotheken (Abschnitt 4.2) aufgeführt. In Abschnitt 4.3 wird die Vorgehensweise bei der Implementierung beschrieben, gefolgt von den zu versendenden Nachrichtentypen (Abschnitt 4.4) und dem grundlegenden Aufbau in Abschnitt 4.5. Außerdem wird dargestellt wie die Kryptografie (Abschnitt 4.6) umgesetzt wurde und es werden weitere Device- und Server-Aktoren vorgestellt (Abschnitt 4.7). Abschließend wird in Abschnitt 4.8 die Umsetzung des Watchdog-Timer erläutert.

Der Code der Implementierung ist der gebundenen Ausgabe der Masterarbeit auf CD als .zip-Datei beigefügt. In Anhang A.3 wird aufgeführt, welche Dateien und Ordner in der .zip-Datei enthalten sind. In der `README.md` sowie im Anhang unter A.1 befindet sich eine Anleitung zum Installieren der Software mithilfe der .zip-Datei.

Die Proof-of-Concept-Implementierung des Watchdog-Timer-Protokolls beinhaltet verschiedene Ordner und Python-Skripte. Den Hauptteil der Implementierung stellen die drei Dateien `app.py`, `device.py` und `server.py` dar, die den Aufbau und die Funktionen des Aktorsystems und ihrer Aktoren beinhalten (Abschnitte 4.5, 4.7.1 und 4.7.2). Des Weiteren werden Module erstellt, die die Implementierung des Protokolls erleichtern. Die Implementierung orientiert sich stark an der Konzeption (Kapitel 3) und den dort beschriebenen Szenarien (Abschnitt 3.1.4). In diesem Kapitel werden die Abweichungen der Implementierung zur Konzeption und relevante Implementierungsdetails beschrieben.

4.1. Systemanforderungen

Es wird davon ausgegangen, dass klassische kryptografische Primitive gebrochen wurden oder in Zukunft gebrochen werden können (vgl. Abschnitt 1.1). Daher werden in diesem Abschnitt die Anforderungen an die Implementierung und die dazugehörige Hardware gestellt, die das neu konzipierte Protokoll erfüllen muss, um sich vor potenziellen Angreifern zu schützen. Aus Abschnitt 3.1.7 zum abgewandelten Watchdog-Timer-Protokoll gehen die funktionalen Anforderungen hervor. Ein Teil der nicht-funktionalen Anforderungen wurden in Absprache mit den Referenten der Masterarbeit festgelegt. Die weiteren nicht-funktionalen Anforderungen wurden durch die Protokolle CIDER [3] und Lazarus [2], sowie durch die Einschränkungen aufgrund der gewählten Programmiersprache, vorgegeben.

4.1.1. Funktionale Anforderungen

Das System soll aus einem Server und einem IoT-Gerät bestehen, die mithilfe des Watchdog-Timer-Protokolls miteinander kommunizieren, das in Kapitel 3 neu konzipiert wurde. Die in Abschnitt 3.1.5 erläuterten Nachrichtentypen sollen im Protokoll verwendet werden und deren jeweilige Schutzziele müssen erfüllt werden. Das Protokoll soll alle Szenarien durchführen können, die in Abschnitt 3.1.4 herausgearbeitet wurden. Zum Sicherstellen der Integrität der Nachrichten soll mindestens die Hashfunktion SHA256 genutzt werden. Um das Protokoll auch im Zeitalter von Quantencomputern einsetzen zu können, soll die Authentizität der Kommunikationspartner im Protokoll durch quantensichere digitale Signaturalgorithmen sichergestellt werden. Das Ziel soll es sein, den klassischen Algorithmus so zu implementieren, dass er leicht gegen den quantensicheren Algorithmus ausgetauscht werden kann.

4.1.2. Nicht-funktionale Anforderungen

Das Protokoll soll in der Skriptsprache Python mithilfe des Aktor-Frameworks Thespian (vgl. Abschnitt 2.2) in Software implementiert werden. Die möglichen zu verwendenden Bibliotheken werden daher auf Python-Packages und Wrapper-Bibliotheken beschränkt (vgl. Abschnitt 4.2). Das Design des Watchdog-Timer-Protokolls soll generisch umgesetzt werden, sodass Aktoren und deren Aufgaben leicht hinzugefügt oder entfernt werden können. Die Schlüsselverteilung soll nicht als Bestandteil des Protokolls angesehen werden.

4.2. Verwendete Frameworks und Bibliotheken

Das Proof-of-Concept des Watchdog-Timer-Protokolls wurde mithilfe des Aktor-Frameworks und Python-Packages `thespian` (Abschnitt 2.2) ausschließlich in Software in einer virtuellen Python-Umgebung implementiert. Vor- und Nachteile der Skriptsprache und des Aktor-Frameworks werden in Abschnitt 6.1.2 diskutiert.

Für CIDER gibt es laut aktuellem Stand keinen Prototyp, der online verfügbar ist. Laut Huber *et al.* [2] verwendet Lazarus zum Teil die kryptografische Bibliothek RIoT [75], um kryptografische Primitive wie die Verifizierung des Boottickets umzusetzen. Der Lazarus-Prototyp [35] wurde in C implementiert und arbeitet mit der C-Bibliothek `mbedtls` [76]. Um die klassischen Signaturalgorithmen zu implementieren, wird die Wrapper-Bibliothekt `python-mbedtls` [77] verwendet. In Abschnitt 3.2 wurde erläutert, welche PQC-Algorithmen im weiteren Verlauf der Arbeit betrachtet und bei den Messungen in Kapitel 5 verwendet werden. In der Wrapper-Bibliothek `liboqs-python` sind alle zu testenden Algorithmen (vgl. 3.2.2 und 2.4.2) vorhanden: Dilithium, Falcon und SPHINCS+.

4.3. Vorgehensweise bei der Implementierung

Aus den möglichen Szenarien und ihren Sequenzdiagrammen (Abschnitt 3.1.4) wurde ein Proof-of-Concept implementiert. Die Aktoren orientieren sich an den Klassen, die in Abschnitt 3.1.4 beschrieben wurden.

Als Erstes wurden die AWDT-Schnittstellen (`AWDT_Init`, `AWDT_GetNonce`, `AWDT_PutTicket`), die im Lazarus-System [2] verwendet werden, sinngemäß mithilfe des Python-Aktor-Frameworks Thespian (Abschnitt 2.2) umgesetzt. Die Implementierung der AWDT-Schnittstellen (`AWDT_Init`, `AWDT_GetNonce`, `AWDT_PutTicket`) diente dem Verständnis über Anwendung des Watchdog-Timers im Aktorenmodell und innerhalb des Watchdog-Timer-Protokolls.

Anschließend wurden die Aktoren mit `print`-Statements nacheinander ausgeführt, damit der Nachrichtenfluss nachvollzogen werden konnte. Die `print`-Statements dienten dem Verständnis über die Aktionen, die ein Aktor ausführt, sowie den Versand der Nachrichten zwischen den Aktoren. Diese Implementierung wurde durch weitere Aktoren (vgl. Abschnitt 3.1.4) ergänzt, um das gesamte System nachzustellen. Anschließend wurden anstelle der `print`-Statements die jeweiligen Aufgaben der Aktoren implementiert. Die Aktoren versendeten zunächst keine signierten Nachrichten, sondern Klartext-Strings. Des Weiteren wurden die verschiedenen Nachrichtentypen (vgl. Abschnitt 3.1.5) implementiert und das neu konzipierte, zunächst jedoch klassische, Watchdog-Timer-Protokoll wurde umgesetzt. Dafür wurden die Aktoren `Signer` und `Verifier` mit einem klassischen Signaturverfahren der Bibliothek `python-mbedtls` ausgestattet, sodass die Nachrichten, die zwischen dem Gerät und dem Server versendet werden, vom Sender signiert und vom Empfänger verifiziert werden. Als Nächstes wurden die Funktionen `sign` und `verify` der Aktoren `Signer` und `Verifier` durch verschiedene quantensichere Signaturverfahren der Bibliothek `liboqs-python` ausgetauscht.

4.4. Nachrichtentypen

Die folgenden Nachrichtentypen wurden implementiert (vgl. Abschnitt 3.1.5):

- `Addresses` (enthält die `ActorAddress` von `Device` und `Server`)
- `Message` (enthält `Addresses` und einen der nachfolgenden Nachrichtentypen)
- `BootTicket`
- `Update`
- `DefTicket` (kurz für Deferralticket)
- `Request`
- `MeasuredData`

Die Noncen vom Typ String wurden nicht als alleiniger Nachrichtentyp implementiert, sondern als ein Attribut der Nachrichtentypen `BootTicket`, `DefTicket` und `Request`, die ausschließlich die Datentypen Integer und String enthalten. In der Proof-of-Concepts ist der Inhalt des Update-Strings nur `"update_{timestamp}"` und benötigt somit kaum Speicherplatz.

Die Datenstruktur `Message` stellt den primär verwendeten Typ, der zwischen den Aktoren versendeten Nachrichten dar. Die Attribute von `Message` sind in Tabelle 4.1 dargestellt. `Message`-Nachrichten werden zwischen den Aktoren und somit auch zwischen `Device` und `Server` hin- und hergesendet.

Attribut	Beschreibung	Startwert
`addresses`	Beinhaltet die Aktoradressen von `Device` und `Server`.	`Addresses(server_addr, device_addr)`
`sequence_list`	Gibt den Nachrichtenfluss der versendeten `Message` seit dem Starten des `TopLevelActor`s an.	`["boot"]` oder `["update"]`
`signature`	Signatur der Nachricht bzw. des Payloads `mdata`	`""`
`crypto`	Art des zu verwendenden digitalen Signaturalgorithmus	`pqc` oder `classic`
`variant`	Zu verwendender digitaler Signaturalgorithmus	z.B. `Falcon-512`
`scenario`	Szenario, das für die Durchführung einer Messung ausgeführt werden soll.	`None`, `1`, `2`, `3`, `4`, `5`, `6`, `7` oder `8`
`hash_algo`	Bei klassischer Kryptografie zu verwendender Hashalgorithmus zum Erstellen des Hashes vom Payload `mdata` für die Signatur	`sha256`, `sha384` oder `sha512`
`mdata`	Daten der Nachricht, auch *Payload* genannt	`None`

Tabelle 4.1.: Startwerte der Attribute des Nachrichtentyps Message mit Beschreibung. Addresses ist ein Datentyp, der aus zwei ActorAddress-Datentypen zusammengesetzt ist.

`Message`s können von jedem Aktor verändert werden, indem die Inhalte überschrieben werden. Ein Beispiel ist das Senden eines `Request`s an den Server. Der `Verifier` verifiziert die Signatur der Anfrage und der `BootticketGenerator` ersetzt in `Message.mdata` die Anfrage `Request` mit dem generierten `BootTicket`. Die `Message` beinhaltet außerdem Attribute, die als Zustandsvariablen über den zu verwendenden Signaturalgorithmus und im Fall eines klassischen Signaturalgorithmus ggf. den darin zu verwendenden Hashalgorithmus agieren. Die Attribute heißen `crypto`, `variant` und `hash_algo` (vgl. Tabelle 4.1) und werden ent-

weder durch Parametereingaben vom Anwender (vgl. Abbildung 4.1 und Tabelle 4.2) oder durch die Standardeinstellungen beim Starten der Anwendung gewählt. Das Programm wendet aufgrund der Inhalte der Attribute den gewünschten Signaturalgorithmus an. Die `sequence_list` wird von jedem Aktor durch seinen Aktornamen ergänzt, sodass der Versand der Nachrichten zwischen den Aktoren später nachvollzogen werden kann. Die Wahl des `scenario` ist nur bei der Durchführung von Messungen wie CPU-Zyklen relevant (siehe auch Kapitel 5). `mdata` kann nach dem Start einen der Nachrichtentypen `BootTicket`, `Update`, `DefTicket`, `Request` oder `MeasuredData` als Datentyp haben.

Die Nachrichten des Typs `Message`, die zwischen den Aktoren `Device` und `Server` ausgetauscht werden, müssen in der realen Welt übers Internet versendet werden. Die Kommunikationspartner nutzen in der realen Welt nicht zwangsläufig jeweils die gleiche Technologie. Um Daten zum Versenden einheitlich zu gestalten, wird daher das Standard-Dateiformat JavaScript Object Notation (JSON) verwendet. Die darin gespeicherten Datenstrukturen dürfen ausschließlich serialisierbare Datentypen enthalten.

In einer `Message` werden allerdings auch die Adressen der Aktoren `Device` und `Server` als Datentyp `Addresses` gespeichert. Die Adressen haben im Framework den nicht serialisierbaren Typ `ActorAddress`. Um dieses Problem zu umgehen, werden die Nachrichten zwischen den `Device`- und `Server`-Aktoren weiterhin als `Message`s versendet.

Für die Nachrichten, die zwischen `Device` und `Server` ausgetauscht werden, wird in der Implementierung ein Tupel erstellt: `(Message, Addresses)`. Die Datentypen in `Message` werden alle so angepasst, dass sie bei Bedarf mit Hilfsfunktionen serialisiert oder deserialisiert werden können, um als JSON-String über das Netzwerk versendet werden zu können. Dabei wird die `ActorAddress` wird als String dargestellt. Das `Addresses` im Tupel enthält die beiden Aktoradressen von `Device` und `Server`, da der Empfänger diese noch benötigt und sie nicht von einem String wieder in eine `ActorAddress` umgewandelt werden können. Um eine teilweise reale Datenübertragung widerzuspiegeln, wird der restliche Teil von `Message` beim Sender serialisiert und beim Empfänger deserialisiert. Die Strings in `Message.addresses` werden durch die `Addresses` vom Typ `ActorAddress` aus dem Tupel ersetzt. So können die Aktoren der Empfänger-Seite wieder mithilfe der Aktoradressen und dem ursprünglichen Typ `Message` kommunizieren.

4.5. Grundlegender Aufbau der Implementierung

Das Skript `app.py` dient als Startprogramm für die Implementierung bzw. die Applikation (vgl. Abbildung 4.1). Das Startprogramm ist ein Python-Skript, das in einer virtuellen Umgebung unter Angabe von Parametern als Argumente gestartet wird. Die Argumente werden mithilfe eines Argument-Parsers implementiert und in Variablen gespeichert.

Die Parameter, die der Argument-Parser entgegennehmen kann, sind in Tabelle 4.2 aufgeführt. Neben den festgelegten Standardparametern können auch abweichende Parameter,

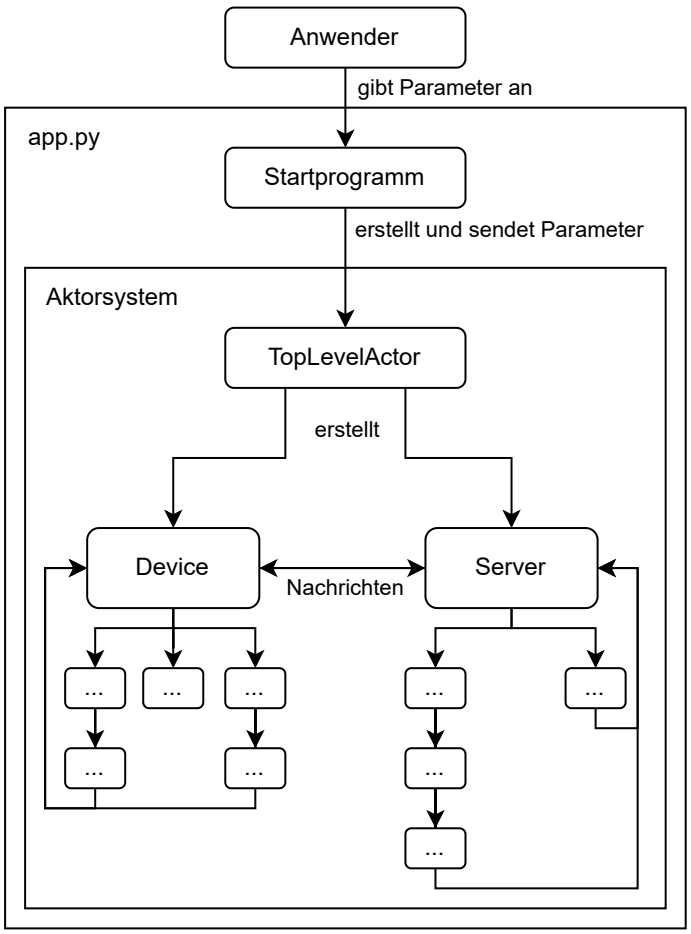

Abbildung 4.1.: Aufbau der Implementierung

wie z.B. die aus der Spalte *Optionen* eingegeben werden. `app.py` kann noch weitere Parameter entgegennehmen, die jedoch nur bei der Durchführung von Messungen relevant sind (siehe Abschnitt 5.3.3 und Tabelle 5.2).

Neben der Entgegennahme der Parameter werden in der Datei `settings.py` Konstanten wie z.B. Speicherpfade und Dateinamen definiert, auf die alle anderen Dateien und Skripte Zugriff haben. Des Weiteren werden dort die Intervalle für die `WakeupMessage`s festgelegt, die den zeitlichen Verlauf des Watchdog-Timer-Protokolls beeinflussen (vgl. Abschnitt 2.2).

Um das Protokoll z.B. unter Verwendung des quantensicheren Algorithmus Falcon-1024 für die Erstellung und Verifizierung der Signatur auszuführen, erfolgt die folgende Eingabe der Parameter.

```
python3 app.py --action=boot --crypto=pqc --variant=Falcon-1024
```

Parameter	Beschreibung	Standardwert	Optionen
`action`	Booten des Geräts oder Server generiert ein Update	`boot`	`boot`, `update`
`crypto`	Art der zu verwendenden Kryptografie	`pqc`	`classic`, `pqc`
`variant`	Zu verwendender Algorithmus	`Falcon-512`	andere Algorithmen (vgl. Tabelle 5.3)
`hash`	Zu verwendender Hashalgorithmus beim Signieren bzw. Verifizieren mit einem klassischen Verfahren	`sha256`	`sha256`, `sha384`, `sha512`
`scenario`	Für die Messreihe durchzuführendes Szenario (vgl. Abschnitt 3.1.4)	`None`	`1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`

Tabelle 4.2.: Auswahl an optionalen Parametern für den Argument-Parser

Wird das Startskript ohne Parameter gestartet, wird das klassische Watchdog-Timer-Protokoll ohne bestimmtes Szenario unter Verwendung des secp256r1-Algorithmus mit der Hashfunktion SHA256 angewandt.

Durch die Abgrenzungen (Abschnitte 1.4.3 und 3.1.1) und Systemanforderungen (Abschnitt 4.1) wird festgelegt, dass sich bei der Konzeption und Implementierung des Watchdog-Timer-Protokolls auf ein Gerät und einen Server beschränkt wird. Daher wird bei der Umsetzung des Protokolls ein Aktorsystem entwickelt, das jeweils nur ein Gerät und einen Server erstellt.

Der Aufbau der Anwendung mit dem Protokoll als Thespian-Aktorsystem wird in Abbildung 4.1 dargestellt. Um das Gesamtsystem (Abbildung 4.1) zu starten, wird vom Startprogramm zunächst eine Thespian-Aktorsystem-Instanz auf dem Hostsystem initialisiert, auf dem das Watchdog-Timer-Protokoll genutzt werden soll.

Als `ActorSystemBase` der Instanz `ActorSystem` wird `multiprocTCPBase` gewählt, sodass alle Nachrichten zwischen den Aktoren per Transmission Control Protocol (TCP) versendet werden und parallel mehrere Aktoren Aufgaben ausführen können. Mit dieser Basis wird ein persistentes Aktorsystem erstellt, das auch nach der Beendigung des Startprozesses weiter existiert [53].

Nach der Initialisierung des Aktorsystems, erstellt das Aktorsystem automatisch die Aktor-Instanz `TopLevelActor`, die den höchsten Aktor darstellt und das Protokoll innerhalb des Aktorsystems startet. Der `TopLevelActor` kann als ein sogenannter `ParentActor` bezeichnet werden, da von ihm alle weiteren Aktoren ausgehen.

Als Erstes erstellt der `TopLevelActor` die beiden Instanzen `Device` und `Server` - sogenannte `ChildActor`s. Im Folgenden wird jedoch nicht der `TopLevelActor` als `ParentActor` bezeichnet, sondern nur die Aktoren `Device` und `Server`, die in dieser Implementierung jeweils als Untersysteme des Aktorsystems angesehen werden. Das Aktorsystem kann vom Skript oder vom `TopLevelActor` mit dem Aufruf von `ActorSystem().shutdown()` beendet bzw. heruntergefahren werden. Von allen Aktoren existiert immer nur eine Instanz. Aus diesem Grund kann davon ausgegangen werden, dass immer die Instanz eines Aktors gemeint ist, wenn von einem Aktor geschrieben wird. Aktoren wie `Device`, `Server`, `Timer` und `Sensor` haben jeweils einen globalen Namen, mit dem die existierende Instanz angesprochen wird. Das Aktorsystem erstellt den Aktor mit diesem globalen Namen, wenn er noch nicht im Aktorsystem existiert. So kann verhindert werden, dass eine neue Instanz erstellt wird, wenn bereits eine existiert.

Die Namen der Aktoren werden aus den Namen der Klassen aus Abschnitt 3.1.3 übernommen. Der Zusatz Aktor ist optional, das bedeutet `Signer` und `Signer`-Aktor werden als äquivalent angesehen. Alle Aktornamen werden in diesem Kapitel in der Schriftart Verbatim geschrieben. `Device`- bzw. `Server`-Aktoren werden die Aktoren genannt, die Kind-Aktoren des `Device`- bzw. des `Server`-Aktors sind. Sobald ein Aktor erstellt wurde, können diesem Aktor Nachrichten gesendet werden, die eine bestimmte Aktion in diesem Aktor auslösen. Nachdem der `TopLevelActor` die beiden Aktoren `Device` und `Server` erstellt hat, startet er das Programm, indem er eine initiale Nachricht an das `Device` oder den `Server` sendet.

Der Anwender legt beim Starten von `app.py` fest, mit welchem Signaturalgorithmus das Protokoll durchgeführt werden soll. Die entgegengenommenen Parameter (Tabelle 4.2) werden mithilfe einer sogenannten Startnachricht als Dictionary an den `TopLevelActor` übergeben. Der `TopLevelActor` erstellt zunächst eine `Message` (vgl. Abschnitt 4.4) mit leeren Strings für das Header-Attribut `signature` und füllt die übrigen Attribute mit den Parameterangaben aus der Startnachricht (vgl. Attribute mit Startwerten aus Tabelle 4.1). Der `TopLevelActor` erhält mit der Startnachricht eine Stringvariable, die angibt, an welchen Aktor er die `Message` senden soll. Es gibt zwei Möglichkeiten:

- `"boot"`: `Device` erhält eine `Message` mit `sequence_list=["boot"]`.
- `"update"`: `Server` erhält eine `Message` mit `sequence_list=["send_update"]`.

Für jeden weiteren Aktor, in dem die Nachricht später verarbeitet wird, wird der Name des Aktors in der `sequence_list` als Element angehängt. Diese Liste dient dazu, im Nachhinein die Reihenfolge der angesprochenen Aktoren nachvollziehen zu können.

Jeder Aktor hat den Zweck eine bestimmte Aufgabe zu erledigen oder eine bestimmte Aktion durchzuführen, die aus mehreren Teilen oder Varianten bestehen kann. Die zu erledigende Aufgabe wird durch die Kombination aus Art und Sender der Nachricht bestimmt. Falls bespielsweise der `Device`-Aktor eine Nachricht vom `TopLevelActor` erhält, wird der Bootvorgang gestartet, der anschließend eines der Szenarien 2, 3, 6 oder 7 startet, abhängig davon, welche Datei in der Staging Area des Geräts vorliegt (vgl. Szenarien in Abschnitt 3.1.4). Erhält der `Server`-Aktor eine Nachricht vom `TopLevelActor`, so generiert der `Server`-Aktor ein Update und führt Szenario 8 durch (vgl. Abschnitt 3.1.4). Erhält der

`Device`-Aktor Nachrichten vom `Server` so wird nach der Art der Nachricht entschieden, an welchen Kind-Aktor die Nachricht weitergeleitet wird. Vom `Device`-Aktor kann der `Server`-Aktor verschiedene andere Nachrichten erhalten. Je nachdem welche Nachricht der `Server`-Aktor erhält, erstellt er sich einen `ChildActor`, der dann eine bestimmte Aktion durchführt.

`Server` und `Device` können jeweils nur bestimmte Aktoren erstellen, wie in den Abbildungen 4.2 und 4.3 dargestellt ist. Grund dafür ist, dass der Server nicht die Aufgaben des Geräts ausführen darf und vice versa.

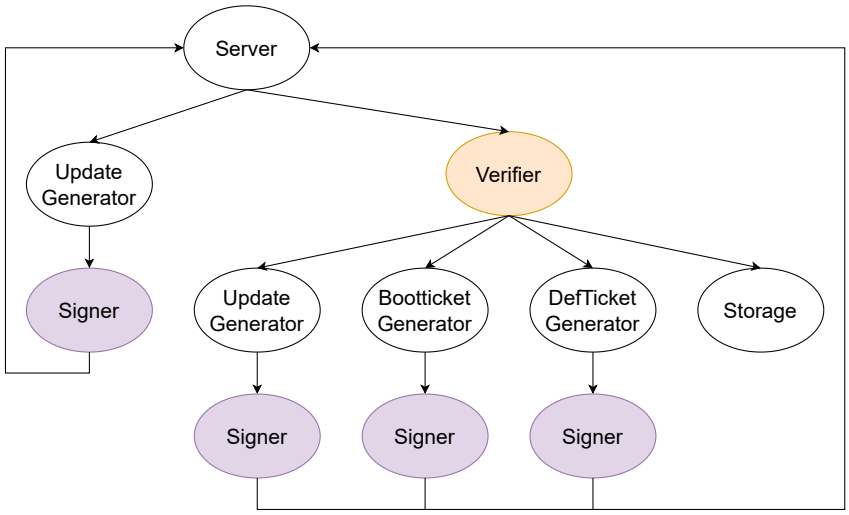

Abbildung 4.2.: Übersicht über die Server-Aktoren, die zeigt, welche Aktoren von welchem Aktor erstellt werden können.

In Abbildung 4.3 haben die Aktoren unterschiedliche Farben, die unterschiedliche Merkmale darstellen. `Timer` und `Sensor` können parallel zu anderen Aktoren laufen und haben immer die gleiche `ActorAddress`, da sie globale Namen haben. `Signer` und `Verifier` werden in `Device` und `Server` jedes Mal neu erstellt, da sie keinen globalen Namen haben (vgl. Abbildungen 4.2 und 4.3).

Für die Kommunikation zwischen den beiden Systemen, die durch die Aktoren `Server` und `Device` dargestellt werden, wurde eine Schnittstelle implementiert, über die jegliche Nachrichten versendet werden. Die Kommunikation zwischen `Device` und `Server` wurde so umgesetzt, dass sie nur durch diese beiden Aktoren möglich ist (vgl. Abbildung 4.1). Alle anderen Aktoren sind so implementiert, dass sie keine Nachricht direkt an beliebige Aktoren des Kommunikationspartners (`Device` oder `Server`) senden können, sondern die Nachricht zuerst an den jeweiligen `ParentActor` gesendet werden muss, damit dieser die Nachricht dann aufbereitet und versendet.
Dies wird auch in den Abbildungen 4.1, 4.3 und 4.2 deutlich. Jeder Pfeil eines Aktors führt am Ende wieder zum `ParentActor`, damit dieser dann eine Nachricht an den Kommunikationspartner senden kann. Grund dafür ist eine bessere Übersicht und die Tatsache, dass die

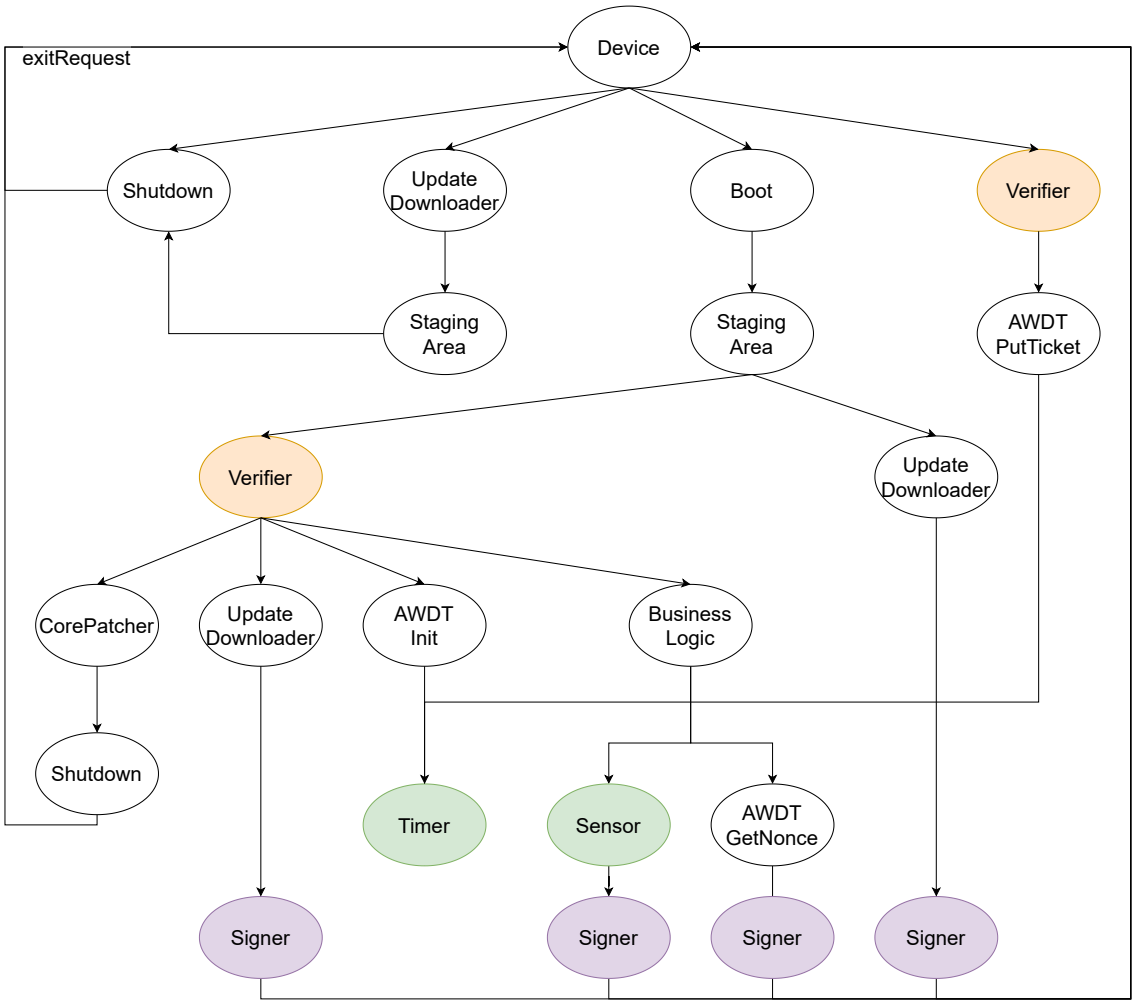

Abbildung 4.3.: Übersicht über die Device-Aktoren, die zeigt, welche Aktoren von welchem Aktor erstellt werden können.

`Message` nicht vor jedem Senden innerhalb des `Device`s oder `Server`s in das JSON-Format umgewandelt werden muss.

Wenn es keinen Pfeil gibt, ist das Szenario mit der Durchführung der Aufgabe des Aktors beendet. Ein Beispiel dafür ist das Speichern von Daten im `Storage` des Servers in Abbildung 4.2. Wenn der `Storage` seine Aufgabe erfüllt hat, beendet er sich selbst und sendet eine `ActorExitRequest` an seinen Sender, damit dieser sich auch beendet. Wenn ein Aktor seine Aufgabe erfüllt hat und die Aufgaben seiner Kinder-Aktoren erfüllt wurden, kann an diese Aktoren eine `ActorExitRequest` gesendet werden, woraufhin der Aktor und alle seine Kinder-Aktoren beendet werden. Diese Nachricht kann der Aktor sich selbst oder anderen Aktoren senden.

Um einen Device-Reset herbeizuführen, wird der Shutdown-Aktor benachrichtigt, der dann eine ActorExitRequest an das Device, den Timer und den Sensor sendet. Durch den Empfang der ActorExitRequest wird das Device beendet und der TopLevelActor wird darüber durch eine ChildActorExited-Nachricht informiert. Durch die ChildActorExited-Nachricht sendet der TopLevelActor sich selbst ein shutdown() [52], woraufhin sich das Aktorsystem mit allen Device- und Server-Aktoren herunterfährt.

In der Theorie muss jedoch nur das Gerät heruntergefahren werden, da der Server nicht kompromittiert wurde. Um einen Reset des Geräts durchzuführen, ohne das gesamte Aktorsystem neu zu starten, kann der Shutdown-Aktor so angepasst werden, dass der Aktor eine Message an das Device sendet anstelle einer ActorExitRequest. Das Device erstellt dann eine neue und leere Message mit den relevanten Parametern und das Device startet sich selbst neu, ohne dass der TopLevelActor benachrichtigt wird.

Alternativ könnte der TopLevelActor eine ChildActorExited-Nachricht vom beendeten Device erhalten, daraufhin ein neues Device erstellen und diesem mit einer neuen Message eine Nachricht (inklusive der neuen ActorAddress) senden, um den Aktor zu resetten.

Um zu garantieren, dass das Skript app.py so lange aktiv ist, wie das Aktorsystem selbst existieren soll, wird eine while-Schleife implementiert. Mithilfe der Schleife wird alle 2 Sekunden überprüft, ob eine bestimmte Dummy-Datei vorhanden ist. Wenn sie nicht mehr vorhanden ist, ist das Aktorsystem beendet worden. Der TopLevelActor erhält eine ActorExitRequest und löscht die Dummy-Datei bevor er aufgrund der ActorExitRequest das gesamte Aktorsystem - inklusive sich selbst - mit dem Aufruf von shutdown() beendet. Da sich die Schleifenbedingung durch das Löschen der Dummy-Datei geändert hat, beendet sich das Aktorsystem, falls es nicht schon vom TopLevelActor beendet wurde. Anschließend wird das Skript beendet und es kann neu ausgeführt werden. Da die while-Schleife in app.py und der TopLevelActor parallel laufen bevor der TopLevelActor bzw. das Aktorsystem beendet wird, ist es möglich, dass entweder der externe Code oder der TopLevelActor das Aktorsystem mit shutdown() beendet.

4.6. Implementierung der Kryptografie

Die Funktionen zum Erstellen von Schlüsselpaaren und Signaturen sowie der Verifizierung von Signaturen sind im Modul crypto zusammengefasst. Dem Modul crypto gehören die drei Objektklassen KeyGen (Abschnitt 4.6.1), SignMessage und VerifyMessage (Abschnitt 4.6.2) an. Bei den Objektklassen SignMessage und VerifyMessage können bei Bedarf noch weitere spezifische sign- und verify-Funktionen generisch hinzugefügt werden, ohne dass am Aktor selbst etwas geändert werden muss.

Die Implementierung wurde so modularisiert, dass sie entweder unter der Verwendung eines klassischen oder eines quantensicheren Algorithmus ausgeführt werden kann. Der verwendenden Signaturalgorithmus wird durch die Parameterangaben --hash, --crypto und --variant beim Starten des Skripts app.py bestimmt (vgl. Abschnitt 4.5). Bei den quantensicheren Signaturalgorithmen werden direkt die darin inkludierten Hashfunktionen,

meist SHAKE256, verwendet und auf die Angabe eines gewünschten Hashalgorithmus wird nicht eingegangen.

4.6.1. Schlüsselerstellung

Die Implementierung des Device Provisionings stellt keine Systemanforderung dar. Da die Schlüsselpaare allerdings für die Verwendung der Funktionen `sign` und `verify` relevant sind, wurde zunächst ein einzelnes Python-Skript `key_generation.py` geschrieben, das mit den Methoden der Klasse `KeyGen` die Schlüsselpaare von Gerät und Server vor dem Ausführen der Anwendung erstellt und verteilt.

Zum Erstellen der Schlüsselpaare kann sowohl ein klassischer als auch ein quantensicherer Algorithmus verwendet werden (vgl. Abschnitt 4.2). Der zu verwendende digitale Signaturalgorithmus wird durch die Parametereingabe beim Ausführung des `key_generation.py`-Skripts mithilfe der Variablen `--crypto` und `--variant` bestimmt. Die Erklärungen zur Parametereingabe sind die gleichen wie beim Startskript (Tabelle 4.2). Bei `--crypto` kann zwischen `classic` oder `pqc` gewählt werden. `--variant` gibt den konkreten Signaturalgorithmus an. Für die Erstellung von Schlüsselpaaren mit klassischer Kryptografie wird wahlweise der Typ `ECC` oder `RSA` (vgl. Abschnitte 2.3.1.3 und 2.3.1.4) der Bibliothek `python-mbedtls` verwendet. Die Bibliothek unterstützt viele verschiedene elliptische Kurven (Abschnitt 2.3.1.4). Um jedoch die Auswahl gering zu halten und die Schlüssel im `.der`-Format speichern zu können, wurde sich bei dieser Implementierung auf die Kurve *secp256r1* beschränkt. Zur quantensicheren Erstellung von Schlüsselpaaren wird die Funktion `signer.generate_keypair` sowie `signer.export_secret_key` der Bibliothek `liboqs-python` verwendet.

Bei der Schlüsselerstellung muss bereits der Algorithmus feststehen, der bei der Erstellung und Verifizierung der Signatur bei der Anwendung im Protokoll verwendet werden soll, da bei allen drei Funktionen der gleiche Algorithmus verwendet werden muss. Ist dies nicht der Fall, kann die Signatur später nicht erstellt bzw. verifiziert werden.

Die Verteilung der Schlüssel wird im Skript `key_generation.py` durch das Speichern der jeweiligen Schlüssel in festgelegte Ordner im Dateisystem des Hosts umgesetzt. Die Speicherorte und Dateinamen der Schlüssel werden in der Datei `settings.py` als Pfade festgelegt und das Skript `key_generation.py` kann darüber auf alle Speicherbereiche des Geräts und des Servers zugreifen.

Nachdem die Schlüsselpaare erstellt wurden, werden die jeweiligen öffentlichen und privaten Schlüssel im `.der`-Format in Dateien, wie z.B. `device_pub_key.der` und `device_priv_key.der`, exportiert und gespeichert. Der öffentliche und der private Schlüssel des Geräts werden im sicheren Speicher des Geräts `device_secure_storage` abgelegt. Der öffentliche und der private Schlüssel des Servers hingegen werden im sicheren Speicher des Servers `server_secure_storage` abgelegt. Außerdem wird der öffentliche Schlüssel des Geräts im Speicher des Servers und der öffentliche Schlüssel des Servers im Speicher des Geräts hinterlegt.

Zusätzlich zum Abspeichern der Schlüssel werden noch die Signer-Details des zur Erstellung verwendeten Algorithmus in einer Datei gespeichert, damit später manuell überprüft werden kann, auf der Basis welchen Commits der Algorithmus verwendet wurde (vgl. Gütekriterien in Abschnitt 6.4.2).

Um das Skript mit z.B. einem klassischen Signaturalgorithmus secp256r1 anstelle des standardmäßig festgelegten Signaturalgorithmus (Falcon-512) auszuführen, erfolgt die folgende Eingabe in der eingerichteten und zu verwendenden virtuellen Python-Umgebung bevor `app.py` ausgeführt werden sollte:

```
python3 key_generation.py --crypto=classic --variant=secp256r1
```

4.6.2. Erstellung und Verifizierung von Signaturen

Die Objektklassen `SignMessage` und `VerifyMessage` implementieren jeweils Funktionen zum Erstellen bzw. Verifizieren einer Signatur. `Signer` und `Verifier` des `Device`s und des `Server`s können nur auf die Ordner zugreifen, die für sie vorgesehen sind (`device_secure_storage` und `server_secure_storage`). Die jeweils zur Erstellung oder Verifizierung der Signatur benötigten Schlüssel werden aus dem jeweiligen Speicherort gelesen und entweder als Variable des Typs `ECC`, `RSA` (bei `python-mbedtls`) oder `Bytes` (bei `liboqs-python`) zur weiteren Verwendung gespeichert.

4.6.2.1. Erstellen einer Signatur

Sowohl im `Device` als auch im `Server` kann ein `Signer`-Aktor initialisiert werden, der dazu da ist eine Signatur zu erstellen. Um Nachrichten digital signieren zu können, wird in der Funktion `sign` der Payload (Abschnitt 4.4) zunächst in einen JSON-Bytestring umgewandelt. In der Implementierung wird als Payload der Inhalt des Attributs `Message.mdata` bezeichnet. Anschließend wird evaluiert, ob ein klassischer oder quantensicherer Algorithmus verwendet werden soll und die benötigten Variablen werden dementsprechend an die jeweilige Signierfunktion weitergegeben: `liboqs_sign` für die Verwendung eines quantensicheren Algorithmus oder `classic_sign` bei der Verwendung eines klassischen Algorithmus. Bei der Methode `classic_sign` gibt es entweder die Möglichkeit mit `ECC` oder mit `RSA` und dem privaten Schlüssel zu signieren.
Der Name des dabei zu verwendende Signaturalgorithmus wird beim Starten des Protokolls durch den Anwender festgelegt und im Header der `Message.variant` gespeichert. Der Name des bei einem klassischen Signaturverfahren zu verwendenden Hashalgorithmus wird ebenfalls beim Starten des Protokolls festgelegt und im Header unter `Message.hash_algo` gespeichert. Wie bereits in Abschnitt 3.2.2 erwähnt, ist die Funktion SHA512 am quantensichersten, wobei bei den aktuellen Möglichkeiten von Quantencomputern auch SHA256 als sicher gilt [74].
Bei der Verwendung eines quantensicheren Signaturalgorithmus wird `liboqs_sign()` mit dem privaten Schlüssel des aktuellen `ParentActor`s durchgeführt. Die Hashfunktion wird

vom quantensicheren Signaturalgorithmus vorgegeben und kann nicht vom Anwender festgelegt werden.

Anschließend wird im Header der `Message` die Signatur unter `Message.signature` gespeichert und der originale Payload `Message.mdata` bleibt erhalten. Der `Signer` leitet die Nachricht dann an den aktuellen `ParentActor` weiter, damit dieser die Nachricht an den Kommunikationspartner senden kann. Nach dem Senden der `Message` an den `ParentActor` beendet sich der `Signer`, da er als Aktor nicht mehr benötigt wird. Der `ParentActor` sendet die `Message` dann an den Kommunikationspartner.

4.6.2.2. Verifizieren einer Signatur

Der `Verifier` von `Device` oder `Server` empfängt eine Nachricht und kann `Update`s, `BootTicket`s, `DefTicket`s oder `Request`s verifizieren. Der `Verifier` wandelt anschließend den Payload dieser Nachricht in einen JSON-Bytestring um und verifiziert die Signatur.

Bei der Methode `classic_verify` gibt es die Möglichkeit die Signatur mit entweder `ECC` oder `RSA` zu verifizieren. Die Signatur wird mithilfe des festgelegten digitalen Signaturalgorithmus und dem öffentlichen Schlüssel des Kommunikationspartners verifiziert. Der Name des dabei zu verwendende Signaturalgorithmus wird beim Starten des Protokolls durch den Anwender festgelegt und im Header unter `Message.variant` gespeichert.

Zusätzlich zur Signaturverifizierung werden beim Empfang eines `BootTicket`s oder `DefTicket`s die empfangene und die im Geät gespeicherte Nonce miteinander verglichen. Erst wenn bestätigt wird, dass die Noncen gleich sind, wird die jeweilige `verify()`-Funktion ausgeführt. Beim Empfang eines `Update`s hingegen wird überprüft, ob die Versionsnummer vom neu empfangenen `Update` genau um eins höher ist als das `Update`, das als letztes implementiert wurde. Im Proof-of-Concept gilt ein `Update` als installiert, wenn es vom `Verifier` verfiziert sowie vom `CorePatcher` „installiert“ und aus der `StagingArea` des Geräts entfernt wurde. In einer realen Anwendung muss gewährleistet sein, dass ein richtiges Update installiert wird.

Die Verifizierung von `Request`s und den Sensordaten `MeasuredData` beruht nur auf der Verifizierung der Signatur und des Hashes, die der `Server` vom `Device` empfangen hat. Im `Request` von einem Gerät ist immer eine Nonce gespeichert, die mit dem `BootTicket` oder `DefTicket` wieder an das `Device` zurückgesendet werden muss, damit der `Verifier` des `Device`s die zuvor mitgesendete Nonce und die neu empfangene Nonce miteinander vergleichen kann, um Freshness bzw. Integrität sicherzustellen.

Um den Fall darzustellen, in dem der `Server` Kenntnis davon hat, dass das `Device` kompromittiert ist, wird im Ordner `server_data_storage` manuell die Datei `compromised.device` erstellt. Der `Verifier` validiert erst die Signatur und überprüft anschließend, ob diese Datei vorliegt. Wenn die Datei nicht vorliegt, beauftragt der `Verifier` den jeweiligen Generator mit der Erstellung einer Nachricht. Liegt die Datei vor, wird keine Nachricht generiert und der `Timer` des `Device`s läuft nach Ablauf der zu Beginn festgelegten Zeit ab.

4.7. Aktoren

Die Aktoren werden in zwei Aktorenklassen aufgeteilt: `Device`-Aktoren (Abschnitt 4.7.1) und `Device`-Aktoren (Abschnitt 4.7.2). Beide Aktorklassen haben jeweils einen `Signer` und einen `Verifier`, die in der Implementierung ähnliche, aber unterschiedliche Aufgaben durchführen, die spezifisch für den jeweiligen Aktor sind (vgl. Abschnitte 4.6 und 3.1.3).

4.7.1. Device-Aktoren

In der Datei `device.py` sind alle Aktoren implementiert, die Aufgaben erledigen sollen, die vom Gerät ausgeführt werden müssen, sowie der Aktor `Device` selbst. In Abbildung 4.3 werden die `Device`-Aktoren mit ihren jeweiligen `ParentActor`s dargestellt.

4.7.1.1. Staging Area

Ein wichtiger `Device`-Aktor ist die `StagingArea`, die auf den Ordner `device_staging_area` zugreifen kann. Der `Server` sendet `Update`s oder `BootTicket`s an das `Device`, das die Nachrichten direkt an die `StagingArea` weiterleitet und mithilfe des Dateisystems in dem dafür vorgesehenen Ordner speichert.

Die `StagingArea` ist so implementiert, dass maximal ein `Update` und maximal ein `BootTicket` vorliegen kann. Wird z.B. ein `Update` in der `StagingArea` gespeichert, obwohl dort bereits eines vorliegt, wird das vorhandene `Update` vom neuen `Update` überschrieben. Das Gleiche gilt für eine `BootTicket`-Datei. Der Fall, dass mehrere Updates vorliegen, wird nicht berücksichtigt.

Invalide `Update`s oder `BootTicket`s können vom `Verifier` aus der `StagingArea` entfernt werden, sowie erfolgreich installierte `Update`s vom `CorePatcher` entfernt werden können. Das `BootTicket` kann von der `BusinessLogic` nach erfolgreichem Boot aus dem Ordner der `StagingArea` entfernt werden. Nachdem ein `Update` oder `BootTicket` in der `StagingArea` gespeichert wurde, wird das Gerät resettet (siehe Abschnitte 4.5 und 4.8).

4.7.1.2. Update Downloader

Der `UpdateDownloader` kann `Request`s für die Nachrichtentypen `Update`, `BootTicket` oder `DefTicket` erstellen. Es wird davon ausgegangen, dass wenn der `UpdateDownloader` ein `Request` sendet, der `Server` antwortet. Es wurde somit kein Timeout implementiert, um den Fall abzufangen, in dem der Server nicht antwortet. Wenn der Server zu lange nicht antwortet, kann er keine `DefTicket`s senden. Der `Timer` läuft irgendwann ab und resettet das Gerät und - im Falle der Proof-of-Concept-Implementierung - das gesamte Aktorsystem. Sollte der Server runtergefahren worden sein, wird er durch einen Neustart des Aktorsystems erneut hochgefahren und kann wieder Nachrichten an das `Device` senden.

4.7.1.3. Business-Logik

Die `BusinessLogic` kann unterschiedliche Funktionen ausführen. In dieser Proof-of-Concept-Implementierung handelt es sich um einen `Sensor`, der einen gewissen Wert messen und diesen Wert zur weiteren Verarbeitung an den `Server` senden soll.

Eine weitere Aufgabe der `BusinessLogic` ist das Anfordern einer Nonce mithilfe des Aktors `AWDT_GetNonce` und damit das Erstellen einer `DefTicket-Request`, die vom `UpdateDownloader` erstellt wird.

Alle 10 Sekunden soll eine `DefTicket-Request` an den `Server` gesendet werden. Die wiederholte Erstellung der `Request` wird durch eine `WakeupMessage` von der `BusinessLogic` an sich selbst realisiert. Das Zeitintervall von 10 Sekunden wird in der Datei `settings.py` festgelegt.

4.7.1.4. Sensor

Der `Sensor` ist ein Aktor, der zunächst von der `BusinessLogic` gestartet wird und der alle 10 Sekunden einen Wert messen und ihn an den `Signer` senden soll. Als Beispiel für einen Messwert wird hier eine zufällige Zahl aus Integern zwischen 4 und 14 ausgewählt und symbolisch als Messwert an den `Server` gesendet. Die wiederholte Aufnahme eines Messwerts wird durch eine `WakeupMessage` vom `Sensor` an sich selbst realisiert. Das Zeitintervall von 10 Sekunden wird in der Datei `settings.py` festgelegt.

4.7.1.5. Shutdown

Der `Shutdown`-Aktor beendet das `Device`, den `Timer` und den `Sensor`, da diese sonst weiterhin parallel laufen würden. Das `Device` kann einen Shutdown von sich selbst hervorrufen, wenn z.B. der `Timer` abgelaufen ist, wird dem Aktor `Shutdown` eine Nachricht gesendet. Der `Shutdown`-Aktor sendet eine `ActorExitRequest` an den `Device`-Aktor.

4.7.2. Server-Aktoren

In der Datei `server.py` sind alle Aktoren implementiert, die die Aufgaben haben, die vom Server ausgeführt werden müssen, sowie der Aktor `Server` selbst. In Abbildung 4.2 werden die `Server`-Aktoren mit ihren jeweiligen `ParentActor`s dargestellt.

Die Aktoren `BootticketGenerator`, `UpdateGenerator` und `DefticketGenerator` implementieren jeweils eine Funktion, die den jeweiligen Nachrichtentyp erstellt und in `Message.mdata` zum weiteren Versand speichert. Das jeweils generierte `BootTicket` oder `Update` wird sicherheitshalber noch zur späteren Referenz im Speicher des Servers abgelegt.

Der Fall, dass der Server außerplanmäßig nicht mehr zur Verfügung steht, wird nicht berücksichtigt.

4.7.2.1. Storage

Im Aktor `Storage` wird die Funktion `data_processing()` aufgerufen. In dieser Implementierung wird darin der empfangene Messwert des `Sensor`s mit dem Zeitstempel in eine Datei im Ordner `server_data_storage` gespeichert. Der Speicherpfad für die Sensordaten `MeasuredData` wurde wiederum in der Datei `settings.py` festgelegt. Theoretisch könnten an dieser Stelle weitere Funktionen zur Datenverarbeitung ergänzt werden.

Wenn die Verarbeitung des Nachrichtentyps `MeasuredData` erfolgt ist, beendet der `Storage`-Aktor sich selbst.

4.8. Implementierung des Watchdog-Timers

Der Watchdog-Timer wird mithilfe des Aktors `Timer` als `ChildActor` der `Device`-Instanz implementiert. Der Aktor hat eine Methode zum Empfangen von Nachrichten sowie zwei weitere Methoden zum Ausführen seiner Funktionalität.

`Timer` wird über drei mögliche Schnittstellen (`AWDT_*`), die jeweils als Aktor dargestellt werden, angesprochen. Der Aktor `AWDT_Init` wird erstellt, damit dieser den Aktor `Timer` initialisieren kann. Der `Timer` erhält eine `Message` mit einem zuvor verifizierten `BootTicket` als Payload. Die Zeit bis zum Reset des Geräts wurde in der Implementierung als Startzeit bezeichnet und ist im `BootTicket` gespeichert. Der Startwert für den `Timer` wird bei der `BootTicket`-Generierung in der Datei `generate_objects.py` festgelegt und über das `BootTicket` an den `Timer` weitergegeben. Die im `BootTicket` gespeicherte Zeit von 25 Sekunden - festgelegt in der Datei `settings.py` - stellt die Zeit dar, die bis zu einem Reset maximal verstreichen darf. Um die Zeit des Resets festzulegen, wird auf den aktuellen Zeitstempel diese Startzeit aufaddiert. Zum Überprüfen, ob die Zeit des Timers abgelaufen ist, fragt sich der Timer alle 2 Sekunden mithilfe einer `WakeupMessage`, ob die Zeit des Resets bereits eingetreten ist. Dafür wurde die Funktion `check_countdown` implementiert, die überprüft, ob die Differenz zwischen der Resetzeit und der aktuellen Zeit negativ ist. Falls die Differenz negativ ist, wird der `Shutdown`-Aktor benachrichtigt, der alle Aktoren im Aktorsystem beendet.

Mit dem Aktor `AWDT_GetNonce`, der von der `BusinessLogic` aus erstellt wird, wird eine `DefTicket`-Anfrage mit Nonce generiert. Um die Zeit des Resets mit einem verifizierten `DefTicket` hinauszögern zu können, wurde die Funktion `defer_countdown` implementiert. `AWDT_PutTicket` übergibt das vom `Server` generierte `DefTicket` an den `Timer`. Die Zeit des Resets wird neu festgelegt mit der Zeit in Sekunden, die im `DefTicket` gespeichert ist. Dafür wird die Zeit aus dem `DefTicket` zum aktuellen Zeitstempel hinzuaddiert.

Um den Watchdog-Timer quantensicher zu machen, wurde darauf geachtet, dass der `Timer`-Aktor nur Nachrichten von den AWDT-Schnittstellen-Funktionen `AWDT_Init` und `AWDT_PutTicket` empfangen kann. Diese wiederum wurden so implementiert, dass sie nur vom `Verifier` verifizierte `BootTicket`s oder `DefTicket`s mit `AWDT_PutTicket` an den

`Timer`-Aktor weiterleiten können. Somit ist gewährleistet, dass der `Timer` nur dann initialisiert wird oder das Timeout aufgeschoben werden kann, wenn ein verifiziertes `BootTicket` oder `DefTicket` vorliegt.

Die Umsetzung des Resets, das durch ein Timeout des Watchdog-Timers oder dem Speichern eines Nachrichtentyps in der `StagingArea` hervorgerufen wird, wurde bereits in Abschnitt 4.5 erläutert.

4.9. Implementierung von Hilfsfunktionen

Es wurden verschiedene Funktionen implementiert, um Codedoppelungen auszulagern und somit Codezeilen in den Aktor-Klassen zu reduzieren.

- Eine Funktion zum Überprüfen, von welchem Aktor eine Nachricht gesendet wurde. Dies wird anhand der Aktor-Namen, die in einer sogenannten Sequenzliste gespeichert werden, nachvollzogen. So kann an jeder Stelle im Code der Weg, den eine Nachricht abgelaufen ist, nachvollzogen werden.
- Verschiedene Funktionen zum Speichern und Lesen unterschiedlicher Datentypen.
- Funktionen zum Lesen aus Dateien und speichern in eine Variable.
- Ordner können erstellt werden, wenn sie noch nicht existieren.
- Das Lesen von Schlüsseln aus dem `.der`-Format heraus in Formate wie `ECC`, `RSA` oder `Bytes` wurde außerdem implementiert, damit diese direkt als Parameter für die Signier- und Verifizierfunktionen verwendet werden können.
- Eine Klasse zum Generieren unterschiedlicher Objekte bzw. Nachrichtentypen erstellt, jeweils aufgeteilt in Nachrichtentypen, die nur vom Gerät und solche, die nur vom Server erstellt werden dürfen.
- Eine andere Klasse beschreibt die Datentypen, die die Klassenvariablen der jeweiligen Nachrichtentypen für den Payload `Message.mdata` annehmen sollen.

5. Durchführung und Auswertung der Messreihen

In diesem Kapitel wird beschrieben unter welchen Bedingungen die Messreihen durchgeführt und ausgewertet wurden. In Abschnitt 5.1 werden die dafür verwendeten Ressourcen und Technologien beschrieben. Anschließend wird aufgeführt inwiefern, die in Kapitel 4 beschriebene Implementierung für die Durchführung der Messungen der Szenarien abgeändert wurde (Abschnitt 5.2). Es wurden zwei Arten von Messskripten entwickelt, mit denen CPU-Zyklus-Messungen durchgeführt werden können (Abschnitt 5.3).

Der Hauptteil dieses Kapitels beschäftigt sich nacheinander jeweils mit der Durchführung und Auswertung der Messreihen. Dafür werden zunächst die Messungen der CPU-Zyklen (Zyklen der Central Processor Unit (CPU)) der Python-Wrapper-Funktionen (Abschnitt 5.4) und anschließend der Szenarien des Watchdog-Timer-Protokolls (Abschnitt 5.5) betrachtet.

5.1. Verwendete Ressourcen und Technologien

Für die Umsetzung und Durchführung der CPU-Zyklus-Messungen wurden sowohl Hardware- als auch Softwarekomponenten (Abschnitte 5.1.1 und 5.1.2) eingesetzt.

5.1.1. Verwendete Hardware

Für die Messungen wurde ein Notebook mit den folgenden Spezifikationen verwendet:

- RAM: 7,7 GiB
- Prozessor: Intel Core i7-5600U CPU @ 2,60 GHz x 4
- Graphik: Mesa Intel HD Graphics 5500 (BDW GT2)
- Betriebssystem: Ubuntu 20.04.5, 64-bit

Das Notebook wurde für die Messungen vom WLAN getrennt und alle anderen Programme wurden geschlossen, um die Durchführungszeit der Messreihen so gering wie möglich zu halten. Des Weiteren wurden die Messskripte (siehe Abschnitt 5.3) direkt vom Terminal aus gestartet.

5.1.2. Verwendete Software und Python-Packages

Die Implementierung sowie die Messskripte wurden auf Ubuntu 20.04 in einer virtuellen Python-Umgebung ausgeführt. In der virtuellen Umgebung wurden u.a. die Python-Packages `hwcounter`, `liboqs-python`, `python-mbedtls` und `thespian` verwendet.

Es wurden Änderungen an der in Kapitel 4 vorgestellten Implementierung vorgenommen, damit die Messungen zu den Szenarien durchgeführt werden konnten (Abschnitt 5.2). Des Weiteren wurden unterschiedliche Messskripte (Abschnitt 5.3) geschrieben und für die Durchführung der Messreihen verwendet (Abschnitte 5.4.1 und 5.5.1). In der Installationsanleitung im Anhang unter A.1 wird beschrieben wie die Implementierung ausgeführt wird.

5.2. Änderungen an der Implementierung zum Durchführen der Messungen

Um die Messungen der CPU-Zyklen einzelner Programmabschnitte bzw. Szenarien des Watchdog-Timer-Protokolls (vgl. Abschnitt 3.1.4) durchführen zu können, wurden Änderungen am Programmcode vorgenommen. Die Skripte `app.py`, `device.py` und `server.py` wurden dahingehend angepasst, dass die Implementierung (Kapitel 4) nun mit der Intention gestartet werden kann, eine CPU-Zyklus-Messung eines angegebenen Szenarios aufzunehmen, um z.B. Startzustände herzustellen. Des Weiteren können die Szenarien des Protokolls ausgeführt werden, ohne dass Messungen aufgenommen werden. Aufgrund der Änderungen wird nun an festgelegten Stellen im `TopLevelActor`, `Timer`, `Storage` und im `Verifier` des `Server`s mithilfe einer Variablen auf das aktuelle Szenario geprüft. Je nachdem welches Szenario vorliegt, wird der Wert, der bis zu diesem Zeitpunkt benötigten CPU-Zyklen, gemessen und gespeichert. Dies kann sowohl ein Startpunkt als auch ein Endpunkt der gewünschten Messung sein.

Die Voraussetzung zum Starten von Messungen und zum Speichern von Messwerten ist, dass einer der beiden optionalen Parameter `saveb` (bei den Szenarien 1, 2, 3, 6 und 7) oder `saveu` (bei Szenario 8) auf `True` gesetzt wird (siehe Abschnitt 5.3.3). Die Messung wird kurz vor dem Senden einer Nachricht an `Device` oder `Server` gestartet und der Startpunkt der Messung - als aktueller CPU-Zyklus-Zählerstand - wird in einer Datei gespeichert.

Für die Messungen wird festgelegt, dass für Szenario 3 (vgl. Abschnitt 3.1.4.4) immer nach dem Initialisieren des `Timer`s (Szenario 3a) und dem Speichern des ersten `Sensor`-Messwerts (Szenario 3b) die jeweilige aktuelle CPU-Zyklus-Anzahl als Endpunkt der Messung gespeichert wird. Mit dem an Szenario 3 anschließenden Szenario 4 (vgl. Abschnitte 3.1.4.4 und 3.1.4.5) wird das Aktorsystem durch den Empfang eines validen `DefTicket`s beendet. So können die CPU-Zeiten für einen Durchlauf gemessen werden und das Startskript kann weitere Messungen mit einem komplett neuen Aktorsystem durchführen. Vorteil ist, dass damit keine möglichen Altlasten mit in die neue Messung übernommen werden und der Startzustand neu hergestellt wird.

Der Programmcode wurde so umgeschrieben, dass immer ein Szenario angegeben wird, das einen Startzustand vorgeben kann bzw. für das Speichern von Messwerten relevant ist. Ein Startzustand kann jedoch nur hergestellt werden, wenn `app.py` vom Messskript aus mit einem Parameter für `scenario` gestartet wird. Wird `app.py` direkt vom Terminal ausgeführt - ohne Messskript - so wird der Standardparameter `scenario=none` gewählt. Die Implementierung wird nun zwar mit dem Parameter gestartet, jedoch wird - wie in Kapitel 4 erläutert - das Protokoll durchlaufen, je nachdem welche Dateien an welchen Speicherorten vorliegen. Es wird nicht geprüft, ob der Startzustand des gewählten Szenarios im Dateisystem vorliegt.
Die Angabe des Szenarios muss nur dann erfolgen, wenn für eine bestimmte Messung der richtige Startzustand hergestellt werden muss und die Messungen mit einem spezifischen Dateinamen gekennzeichnet werden sollen.

Durch das Einfügen szenariospezifischer Messpunkte und das frühzeitige Beenden der Ausführung des Programms bei den Messungen von Szenario 3 und 4, wird der ursprüngliche Protokollablauf der Implementierung beeinflusst (vgl. Abschnitt 5.5). Damit das Programm so durchläuft, wie in Kapitel 4 beschrieben, muss Szenario 3 mit anschließendem Szenario 4 durch das Vorhandensein eines `BootTicket`s in der `StagingArea` getriggert werden, ohne dass auf das mitgesendete Szenario geachtet wird. Dabei würde das Gerät immer wieder Messwerte vom `Sensor` an den `Server` senden und das Timeout des `Timer`s wird durch `DefTicket`s immer wieder neu hinausgezögert.

Für die Messung des Szenario 3 beendet der `Storage`-Aktor das `Device`, um danach das gesamte Aktorsystem herunterzufahren und mit dem Antriggern der neuen Messung wieder ein neues Aktorsystem zu erstellen. Um die in Kapitel 4 beschriebene Implementierung wieder zu erhalten, darf sich der `Storage`-Aktor nur selbst beenden und nicht das Device. Daher muss der Empfänger-Aktor der `ActorExitRequest` von `Device` auf `self.myAddress` angepasst werden. Alternativ kann `scenario=none` und die Speicherung von Messwerten `False` gewählt werden, damit es zu keinem Problem mit den Szenario-Abfragen kommt. Die Zeile `self.send(updated_message.addresses.device_addr, ActorExitRequest())` muss im `Timer`-Aktor entfernt werden, damit das `Device` und somit das Aktorsystem nicht nach dem ersten Erhalt eines Deferraltickets beendet wird.

Prinzipiell muss `app.py` so lange laufen, wie das Aktorsystem selbst existieren soll, damit keine weitere Messung gestartet werden kann, während die vorherige noch läuft (vgl. Abschnitt 4.5). Auch beim einmaligen Ausführen des Protokolls sollte darauf geachtet werden, dass nicht mehr als ein Aktorsystem gleichzeitig läuft. Das Herunterfahren eines Aktorsystems wurde in Abschnitt 4.5 erläutert. Beim Durchführen einer Messung wird der Endpunkt aufgenommen und gespeichert, bevor die Dummy-Datei gelöscht wird, wodurch das Aktorsystem heruntergefahren und das Startskript beendet wird.

Soll die Implementierung wie gewünscht durchlaufen werden und das Deferralticket mehr als nur einmal gefetcht werden, so müssen kleinere Anpassungen vorgenommen werden, sodass bei Szenario auch ein `None` möglich ist und bei Szenario 3 nach dem Speichern des ersten Messwerts nicht der `Device`-Aktor, sondern nur der `Storage`-Aktor beendet wird.

5.3. Aufbau der Mess- und Auswerteskripte

Zunächst werden die Grundlagen der Messskripte erläutert. Anschließend wird in einzelnen Abschnitten auf die jeweiligen Messskripte für die Funktionen (Abschnitt 5.3.1) und Szenarien (Abschnitt 5.3.2) eingegangen. In Abschnitt 5.3.3 werden die Parameter zum Starten der Messskripte aufgeführt. Abschließend werden die Aufgaben der Auswerteskripte für die Messreihen beschrieben (Abschnitt 5.3.4).

Um die CPU-Zyklen der verschiedenen Funktionen (`gen_keypair`, `sign`, `verify`) und Szenarien (vgl. Abschnitt 3.1.4) zu messen, verwenden die Messskripte das Python-Package `hwcounter` mit den Funktionen `count` und `count_end`. Des Weiteren wird die Einheit des Messwerts in der `settings.py` mit `UNIT="cycles"` festgelegt. Die Variablen `START_MEASUREMENT` und `END_MEASUREMENT` werden in der Proof-of-Concept-Implementierung als ausführbare Funktionen festgelegt - eine zum Starten und eine zum Beenden der aktuellen Messung.

Die Festlegung der Messfunktionen in `settings.py` sieht wie folgt aus:

```
from hwcounter import count, count_end
START_MEASUREMENT: Callable[[], float] = count
END_MEASUREMENT: Callable[[], float] = count_end
```

Bei Bedarf kann eine andere Messmethode für die Szenarien anstelle der `hwcounter`-Funktionen eingefügt werden. Zum Beispiel könnten die Start- und Endmessungen nicht die CPU-Zyklen angeben, sondern die Laufzeit in Sekunden (z.B. mit `time.perf_counter`) oder den Speicherplatzverbrauch in Byte.

Die Verwendung im Programmcode ist somit generisch und sieht beispielsweise wie folgt aus:

```
from settings import START_MEASUREMENT, END_MEASUREMENT
start_counter: float = START_MEASUREMENT
pub_key: bytes = signer.generate_keypair()
end_counter: float = END_MEASUREMENT
```

In diesem Codebeispiel wird gemessen wie viele CPU-Zyklen die Funktion `signer.generate_keypair` benötigt, um ein Schlüsselpaar zu erstellen. Die Variablen `START_MEASUREMENT` und `END_MEASUREMENT` werden im Programmcode verwendet, um die Anzahl an CPU-Zyklen jeweils einer der drei Funktionen oder eines ganzen Szenarios zu messen. Die Skripte für die Durchführung der Messreihen der Funktionen müssten manuell angepasst werden, da diese direkt die `hwcounter`-Funktionen nutzen.

5.3.1. Messskript für die CPU-Zyklus-Messungen der Funktionen

Es sollen CPU-Zyklus-Messungen der drei Funktionen:

- `gen_keypair`
- `sign`
- `verify`

durchgeführt werden. Für die Messung der CPU-Zyklen der Funktionen wurden verschiedene Signaturalgorithmen implementiert. Die Messreihen für die CPU-Zyklen der Funktionen unter Verwendung klassischer Algorithmen wurden mit einem Messskript (`mbedtls_fkt_messskript.py`) durchgeführt, das ausschließlich die Funktionen der `python-mbedtls`-Bibliothek implementiert hat. Die Messreihen für die CPU-Zyklen der Funktionen unter Verwendung quantensicherer Algorithmen wurden mit einem Messskript (`liboqs_fkt_messskript.py`) durchgeführt, das ausschließlich die Funktionen der `liboqs-python`-Bibliothek implementiert hat.

Die Messskripte, die die CPU-Zyklus-Zeiten der Python-Wrapper-Funktionen `gen_keypair`, `sign` und `verify` messen, beinhalten jeweils ein Beispielprogramm, bei dem ein Schlüsselpaar erstellt, eine Nachricht signiert und anschließend die Signatur der Nachricht verifiziert wird. Vor der jeweiligen Funktion zur Erstellung der Schlüsselpaare, der Signierfunktion und der Verifizierfunktion wird der Zähler der CPU-Zyklen mit `count()` gestartet und direkt nach den Funktionen wird der jeweilige Zähler der CPU-Zyklen mit `count_end()` beendet. Die Differenz der beiden Messwerte ergibt die Anzahl der CPU-Zyklen der jeweiligen Funktion in Cycles.

Die Start- und Endpunkte der CPU-Zyklus-Messung sowie die Differenz der beiden Werte werden in einer `.csv`-Datei gespeichert. Das genannte Beispielprogramm ist bei beiden Messskripten innerhalb einer Schleife implementiert, sodass die CPU-Zyklus-Messungen mehrmals hintereinander durchgeführt werden können.

5.3.2. Messskript für die CPU-Zyklus-Messungen der Szenarien

Das Watchdog-Timer-Protokoll der Proof-of-Concept-Implementierung kann die nachfolgend genannten Szenarien durchspielen, je nachdem welcher Startzustand nach einem Reset vorliegt.

- Szenario 1: Kein Bootticket vorhanden (vgl. Abschnitt 3.1.4.2)
- Szenario 2: Bootticket vorhanden, aber nicht valide (vgl. Abschnitt 3.1.4.3)
- Szenario 3: Bootticket vorhanden und valide (vgl. Abschnitt 3.1.4.4)
- Szenario 4: Business-Logik mit Deferralticket (vgl. Abschnitt 3.1.4.5)
- Szenario 5: Business-Logik mit invalidem Deferralticket (vgl. Abschnitt 3.1.4.6)

- Szenario 6: Update vorhanden, aber nicht valide (vgl. Abschnitt 3.1.4.7)
- Szenario 7: Update vorhanden und valide (vgl. Abschnitt 3.1.4.8)
- Szenario 8: Server sendet Update an Gerät (vgl. Abschnitt 3.1.4.9)

Die Szenarien 4 und 5 werden nicht gemessen, da bei diesen mehrere Aktoren parallel zueinander Aktionen ausführen, wodurch keine genaue Ablaufreihenfolge garantiert werden kann und somit auch die Messpunkte für Starten und Beenden der Messung variieren können (vgl. Abschnitt 6.2.3).

Um die CPU-Zeiten der Szenarien zu bestimmen, wurde zunächst für jedes Szenario ein Startzustand (Tabelle 5.1) herausgearbeitet, der aus den Beschreibungen der einzelnen Szenarien in Abschnitt 3.1.4 hervorgeht. Ein Startzustand beschreibt den Zustand des Dateisystems auf dem Host, wenn die Implementierung ausgeführt wird. Dazu gehören die vorliegenden Schlüssel in `Device` und `Server` sowie ggf. Dateien wie `BootTicket` oder `Update` in der Staging Area des `Device`s.

	Update nicht verfügbar	Update valide	Update nicht valide
Bootticket nicht verfügbar	(1) Bootticket anfragen	(7) Update installieren	(6) Update neu anfragen
Bootticket valide	(3) Normal booten	(7) Update installieren	(6) Update neu anfragen
Bootticket nicht valide	(2) Bootticket neu anfragen	(7) Update installieren	(6) Update neu anfragen

Tabelle 5.1.: Startzustände der einzelnen Szenarien, die nach einem Geräte-Reset in der Staging Area vorliegen können. Die Nummer vor dem jeweiligen Startzustand gibt das dazugehörige Szenario an (vgl. Abschnitt 3.1.4).

In Tabelle 5.1 werden die Nummern 4 und 5 nicht genannt, da deren Startzustände aus dem Booten in die Business-Logik aufgrund eines validen Boottickets hervorgehen würden. Daher wäre der Startzustand in diesem Fall der Start in die Business-Logik mit Szenario 3 (Abschnitt 3.1.4.2). Szenario 8 (Abschnitt 3.1.4.9) beruht nicht auf den Startzuständen, die in nach einem Geräte-Reset in der Staging Area des Geräts vorliegen, daher ist dieses Szenario in Tabelle 5.1 nicht aufgeführt.

Es wurde eine Funktion implementiert, die Hilfsdateien zum Herstellen der Startzustände (vgl. Tabelle 5.1) generiert. Dabei wird das Skript `key_generation.py` ausführt und anschließend die Implementierung mit einem `"boot"` startet, um eine Nonce und ein `BootTicket` zu generieren. Anschließend wird die Implementierung mit einem `"update"` ausgeführt, um ein `Update` zu generieren. Die dabei generierten Schlüssel und anderen Dateien werden an einem sicheren Ort außerhalb des Systems von `Device` und `Server` mit dem jeweiligen Zusatz `0` gespeichert. Das Gleiche wird wiederholt und die generierten Schlüssel und Dateien mit dem jeweiligen Zusatz `1` gespeichert. Die erstellten Schlüssel und anderen

Dateien dienen im Folgenden dazu, die Startzustände für die Szenarien mit immer den gleichen Schlüsseln und Dateien wiederherstellen zu können.

Das Messskript für die Szenarien selbst besteht aus zwei Teilen. Der erste Teil stellt eine Klasse an Methoden dar, die benötigt werden, um den jeweiligen Startzustand für ein bestimmtes Szenario herzustellen. Der zweite Teil stellt das eigentliche Messskript `messskript_szenarien.py` dar. Diesem Skript werden die Parameter (siehe Abschnitt 5.3.3) übergeben, die zur Durchführung der Messungen benötigt werden. Die Messreihen für die CPU-Zyklen der ausgewählten Szenarien greifen über dieses Messskript auf die geänderte Implementierung (Abschnitt 5.2) mit dem Aktor-Framework Thespian zu. Des Weiteren werden hier zunächst die zuvor genannten Hilfsdateien erstellt. Anschließend wird auf das jeweilige Szenario überprüft, für das der Startzustand hergestellt und die Messung aufgenommen werden soll. Das Herstellen des Startzustands und die Ausführung der Implementierung findet innerhalb einer Schleife statt.

Um eine wiederholbare Messung der Szenarien durchführen zu können, muss vor jeder einzelnen CPU-Zyklus-Messung eines Szenarios, der dazugehörige Startzustand (Tabelle 5.1) mithilfe der genannten Methoden hergestellt werden. Dabei werden alle Dateien aus bestimmten Ordnern entfernt und neue Dateien in bestimmte Ordner kopiert. Die CPU-Zeiten, die zur Herstellung des jeweiligen Startzustands benötigt werden, werden nicht mitgemessen.

Für Szenario 2 (vgl. Abschnitt 3.1.4.3 und Tabelle 5.1) werden folgende Schritte durchgeführt:

1. Löschen aller Dateien und Schlüssel im Speicher von `Device` und `Server`.
2. Kopieren aller Schlüssel mit dem Zusatz 0, sowie der Nonce mit dem Zusatz 0 in den Speicher von `Device` und `Server`.
3. Kopieren des `BootTicket`s mit dem Zusatz 1 in die `StagingArea` des `Device`s.
4. Ausführen von `app.py` mit `saveb` zum Speichern der Messungen.

Bei diesem Szenario werden die Schlüssel und die Nonce - mit dem Zusatz 0 in den Dateinamen - auf die gewünschten Ordner verteilt. Sie werden beim ersten Durchführen der Hilfsdateien erstellt. Die Nonce 0 und das Bootticket 0 lassen sich mit den Schlüsseln 0 verifizieren. Da allerdings das Bootticket 1 vorliegt, das mit Nonce 1 und den Schlüsseln 1 erstellt wurde, kann das Bootticket nicht vom Gerät verifiziert werden und fragt mit einer neuen Nonce ein neues Bootticket beim Server an.

Anschließend werden die genannten Schritte für Szenario 2 erneut durchgeführt, um eine weitere Messung mit den gleichen Voraussetzungen zu erhalten. Für die anderen Szenarien werden die Startzustände ähnlich hergestellt.

Zum Speichern von Headern und Messdaten in `.csv`-Dateien wurden weitere Funktionen implementiert.

5.3.3. Parameter der Messskripte

Die Messskripte für die `python-mbedtls`- und die `liboqs-python`-Funktionen benötigen die jeweils in Tabelle 5.2 aufgeführten Eingabeparameter `number`, `hash` und `variant`. Es können nur, die Algorithmen als `variant` angegeben werden, die auch bei den Messungen verwendet werden (vgl. Tabelle 5.3). Des Weiteren kann bei `hash` nur `sha256`, `sha384` oder `sha512` gewählt werden. `hash` gibt die Länge der Nachricht an, die signiert werden soll. Bei der Signaturerstellung und -verifizierung mit einem klassischen Signaturalgorithmus gibt `hash` außerdem die vor der `sign`- bzw. `verify`-Funktion zu nutzende Hashfunktion an. `action` kann sonst nur `update` entgegennehmen.

Parameter	Beschreibung	Standardwert
`number`	Anzahl an Durchführungen des Skripts	`1`
`hash`	Größe der zu signierenden Nachricht 64 Byte und zu verwendende Hashfunktion bei Verwendung eines klassischen Signatualgorithmus	`sha256`
`variant`	Zu verwendender Algorithmus	klassisch: `secp256r1` quantensicher: `Falcon-512`
`action`	Gerät booten oder Server generiert Update.	`boot`
`saveb`	Speichern der Messwerte für das angegebene Szenario; nur in Kombination mit `-action=boot` möglich	`False`
`saveu`	Speichern der Messwerte für Szenario 8; nur in Kombination mit `-action=update` möglich	`False`
`scenario`	Welches Szenario für die Durchführung der Messreihe verwendet werden soll	`1`
`unit`	Welche Einheit bei den Differenzmessungen gemessen wird	`"cycles"`

Tabelle 5.2.: Optionale Parameter für den Argument-Parser zum Durchführen der Funktions- und Szenario-Messungen, die mithilfe des jeweiligen Messskripts gestartet werden, mit Angabe des Standardwerts.

Ein Beispiel für die Messung der `python-mbedtls`-Funktionen:

```
python3 benchmarking/mbedtls_fkt_messskript.py --number=10000
--variant=rsa2048 --hash=sha512
```

Ein Beispiel für die Messung der `liboqs-python`-Funktionen:

```
python3 benchmarking/liboqs_fkt_messskript.py --number=10000
--variant=Falcon-1024 --hash=sha512
```

Zum Aufnehmen einer Messreihe wird Folgendes ausgeführt:

```
python3 messskript_szenarien.py --number=100 --crypto=pqc
--variant=Falcon-1024 --hash=none --scenario=7
```

Das Messskript `messskript_szenarien.py` nimmt die Eingabeparameter `crypto`, `variant`, `hash`, `scenario` und `number` (Tabelle 5.2) entgegen und ergänzt automatisch die Mess- und Speicheroptionen für das jeweilige Szenario mit:

- `--action=update --saveu --unit=cycles` (für Szenario 8) oder
- `--action=boot --saveb --unit=cycles` (für Szenarien 1, 2, 3, 6 und 7)

Anschließend ruft das Messkript mit allen genannten Parametern die geänderte Implementierung von `app.py` (vgl. Abschnitt 5.2) auf.

5.3.4. Auswerteskripte

Die Auswertung der Messreihen erfolgt unabhängig von den Messskripten und der Implementierung. Für die Auswertung der Messreihen der Funktionen und Szenarien wurde jeweils ein Auswerteskript geschrieben. Zunächst werden aus den aufgenommenen Messreihen der Szenarien die Differenzen von den Start- und Endwerten bestimmt.

Beide Auswerteskripte führen für jede Messreihe die folgenden Berechnungen durch und speichern die Ergebnisse in `.csv`-Dateien:

1. Durchschnittswert der jeweiligen Messreihe mit `numpy.median()`
2. Median der jeweiligen Messreihe mit `numpy.mean()`
3. Standardabweichung der jeweiligen Messreihe mit `np.std()`
4. Kleinster Messwert der jeweiligen Messreihe mit `data.min()`
5. Höchster Messwert der jeweiligen Messreihe mit `data.max()`

In den Tabellen der Auswertungen beschreibt *minimaler Wert*, den kleinsten Wert der Messreihe, wobei *maximaler Wert* der höchste Wert der Messreihe ist.

5.4. Messung der CPU-Zyklen der Python-Wrapper-Funktionen

In diesem Abschnitt wird die Durchführung der CPU-Zyklus-Messungen des Python-Wrapper-Funktionen (Abschnitt 5.4.1) sowie die Auswertung der Messreihen beschrieben (Abschnitt 5.4.2).

5.4.1. Durchführung der Messreihen

In Tabelle 5.3 werden die Parameterkonstellationen aufgeführt, mit denen die CPU-Zyklen der drei Funktionen `gen_keypair`, `sign` und `verify` gemessen wurden. Die Verfahren der klassischen Kryptografie (secp256r1, rsa2048 und rsa4096) wurden mit `python-mbedtls` implementiert; alle anderen Verfahren wurden mit der Bibliothek `liboqs-python` implementiert.

Bei der Verwendung der klassischen Signaturalgorithmen gibt `hash` (vgl. Tabelle 5.2) sowohl die Länge der zu signierenden Nachricht als auch den zu verwendenden Hashalgorithmus an. Bei der Verwendung der quantensicheren Signaturalgorithmen wird `hash` ausschließlich dafür verwendet, die Länge der zu signierenden Nachricht anzugeben. Wird z.B. `sha384` gewählt, so wird in beiden Fällen (klassisch und quantensicher) eine Nachricht mit 96 Bytes erstellt und der `sign`-Funktion übergeben. Bei einem klassischen Algorithmus wird `sha384` als Hashalgorithmus in der `sign`-Funktion gewählt. Bei einem quantensicheren Algorithmus wird `sha384` nicht weiter betrachtet. Die Begriffe Nachrichtenlänge und die in `sign` zu verwendenden SHA2-Funktionen werden nachfolgend äquivalent verwendet.

Die Messungen basieren auf der Annahme einer früheren Version der Implementierung, bei der die Nachricht vor der Signierfunktion zusätzlich gehasht wurde, wodurch die Nachrichten beim Signieren immer die gleiche Länge hatten. Daher variieren die Nachrichtenlängen je nach verwendetem Hashalgorithmus. Es wird die doppelte Länge verwendet, da die Hex-Repräsentation als String gespeichert wird.

- Bei `--hash=sha256` wird eine Nachricht mit der Länge 64 Byte gehasht.
- Bei `--hash=sha384` wird eine Nachricht mit der Länge 96 Byte gehasht.
- Bei `--hash=sha512` wird eine Nachricht mit der Länge 128 Byte gehasht.

Es werden mit unterschiedlichen Nachrichtenlängen Messreihen durchgeführt, um zu testen, ob die Nachrichtenlänge einen Einfluss auf die Signaturfunktion hat. In Tabelle 5.3 werden die durchgeführten Messungen mit den verwendeten Parametern aufgeführt. Jede Messung wurde 10.000 Mal mithilfe des Messskripts (Abschnitt 5.3.1) aufgenommen. Die Messungen wurden überwiegend für alle drei Nachrichtenlängen durchgeführt. Für die speicherplatzoptimierten Parametersets von SPHINCS+ (vgl. Abschnitt 2.4.2.5) wurden weitgehend nur Messungen mit einer Nachrichtenlänge von 128 Byte aufgenommen. Die anderen zwei SHA2-Varianten wurden ausgeschlossen, da sich bereits sehr früh gezeigt hat, dass die CPU-Zyklen, der drei Nachrichtenlängen für die gemessenen Funktionen kaum variieren (vgl. Abschnitt 5.4.2.1).

Verfahren	NIST-Level	Nachrichtenlänge (Bytes)		
		64	96	128
pk.Curve.SECP256R1	-	✓	✓	✓
pk.RSA2048	-	✓	✓	✓
pk.RSA4096	-	✓	✓	✓
Falcon-512	1	✓	✓	✓
Falcon-1024	5	✓	✓	✓
Dilithium3	3	✓	✓	✓
Dilithium5	5	✓	✓	✓
SPHINCS+-SHA256-128f-robust	1	✓	✓	✓
SPHINCS+-SHA256-128s-robust	1	✓	✓	✓
SPHINCS+-SHA256-128f-simple	1	✓	✓	✓
SPHINCS+-SHA256-128s-simple	1	-	-	✓
SPHINCS+-SHA256-192f-robust	3	✓	✓	✓
SPHINCS+-SHA256-192s-robust	3	✓	✓	✓
SPHINCS+-SHA256-192f-simple	3	✓	✓	✓
SPHINCS+-SHA256-192s-simple	3	-	-	✓
SPHINCS+-SHA256-256f-robust	5	✓	✓	✓
SPHINCS+-SHA256-256s-robust	5	-	-	✓
SPHINCS+-SHA256-256f-simple	5	✓	✓	✓
SPHINCS+-SHA256-256s-simple	5	-	-	✓
SPHINCS+-SHAKE256-128f-robust	1	✓	✓	✓
SPHINCS+-SHAKE256-128s-robust	1	-	-	✓
SPHINCS+-SHAKE256-128f-simple	1	✓	✓	✓
SPHINCS+-SHAKE256-128s-simple	1	-	-	✓
SPHINCS+-SHAKE256-192f-robust	3	✓	✓	✓
SPHINCS+-SHAKE256-192s-robust	3	-	-	✓
SPHINCS+-SHAKE256-192f-simple	3	✓	✓	✓
SPHINCS+-SHAKE256-192s-simple	3	-	-	✓
SPHINCS+-SHAKE256-256f-robust	5	✓	✓	✓
SPHINCS+-SHAKE256-256s-robust	5	-	-	✓
SPHINCS+-SHAKE256-256f-simple	5	✓	✓	✓
SPHINCS+-SHAKE256-256s-simple	5	-	-	✓

Tabelle 5.3.: Übersicht über die Parameter für die Messreihen der CPU-Zyklen der Funktionen.

5.4.2. Auswertung der CPU-Zyklen der Python-Wrapper-Funktionen

Die Messreihen wurden mit einem Auswerteskript für die Funktions-Messreihen ausgewertet (vgl. Abschnitt 5.3.4). Die von den quantensicheren Algorithmen angestrebten NIST-Sicherheitslevel wurden bereits in Abschnitt 2.4.2.1 erläutert. Zunächst wird der Einfluss der verwendeten Nachrichtenlängen bei der Signaturerstellung evaluiert (Abschnitt 5.4.2.1). Anschließend werden jeweils die Auswertungen der CPU-Zyklen der drei Funktionen unter

Verwendung der in Tabelle 5.3 aufgeführten klassischen und quantensicheren Algorithmen betrachtet (Abschnitte 5.4.2.2, 5.4.2.3 und 5.4.2.4).

5.4.2.1. Auswertung zu verwendeten Nachrichtenlängen bei der Signaturerstellung

In Tabelle 5.4 sind beispielhaft CPU-Zyklus-Messungen für die Funktion `sign` bei der Verwendung unterschiedlicher Algorithmen dargestellt. Die Spalte *Nachrichtenlänge* gibt die Byte-Anzahl der zu signierenden Nachricht an. Im Fall der Funktionsmessungen wird zuvor `os.urandom(msg_len)` als Zufallsgenerator verwendet, der eine zufällige Nachricht mit entweder 64, 96 oder 128 Byte generiert.

Algorithmus	NIST-Level	Nachrichtenlänge (Bytes)	Mittelwert (Cycles)	Median (Cycles)	Standardabweichung (Cycles)	Minimaler Wert (Cycles)	Maximaler Wert (Cycles)
rsa2048	-	64	10.994.658	10.660.416	1.233.149	10.358.114	21.558.620
		96	10.992.283	10.663.952	1.228.033	10.347.549	30.204.077
		128	11.054.261	10.722.024	1.262.677	10.379.275	35.728.107
Falcon-512	1	64	931.455	832.240	235.690	758.172	4.681.508
		96	964.440	856.600	251.776	782.354	3.511.221
		128	923.573	832.906	219.448	762.288	4.535.037
SPHINCS+-SHAKE256-128f-simple	1	64	104.113.111	102.091.246	8.529.783	92.113.150	217.953.623
		96	104.175.348	102.020.717	9.050.370	92.126.103	208.245.455
		128	103.962.746	101.816.930	8.562.325	92.854.465	199.741.175
Dilithium3	3	64	527.255	437.028	305.286	230.094	3.535.783
		96	529.258	435.745	317.616	228.636	3.306.959
		128	534.011	441.538	317.155	229.369	3.116.232
SPHINCS+-SHA256-256f-robust	5	64	418.124.768	412.413.804	23.453.416	398.700.331	605.686.605
		96	419.695.467	413.179.962	25.918.251	399.020.504	705.388.563
		128	421.200.951	414.557.173	27.133.107	399.228.353	665.385.587

Tabelle 5.4.: Vergleich der Auswertungsergebnisse der CPU-Zyklen aufgrund der verwendeten Nachrichtenlängen bei der Signaturerstellung mit der Funktion `sign`. Ausschnitt der Tabellen A.5, A.6, A.7 und A.8 aus Anhang A.5.2.

Die Mediane und Mittelwerte weichen unter Berücksichtigung aller drei Nachrichtenlängen, die in der Tabelle 5.4 dargestellt sind, zwischen 0,02 und 4,4 Prozent voneinander ab. Bei z.B. bei Falcon-512 (vgl. Tabelle 5.4) ist der Mittelwert der `sha384`-Variante höher bei den `sha256`- und `sha512`-Varianten. Bei SPHINCS+-SHA256-256f-robust hingegen ist der Mittelwert bei der `sha512`-Variante höher als bei den beiden anderen Varianten. Auch bei den Medianen variieren die benötigten CPU-Zyklen der drei Varianten. Für die Standardabweichungen, die minimalen und die maximalen CPU-Zyklen der jeweiligen Messreihen gilt dasselbe. Auch bei den Messungen der Funktion `verify` kann eine Unregelmäßigkeit zwischen den drei Varianten festgestellt werden (vgl. Tabellen A.9, A.10, A.11 und A.12 in Anhang A.5.3). Bei Bedarf finden sich die Messauswertungen aller drei Funktionen mit allen drei Nachrichtenlängen in Anhang A.5.

Beim Messskript wurden jeweils die CPU-Zyklen für die Funktion `gen_keypair` bei anschließender Verwendung unterschiedlich großer Nachrichtenlängen gemessen. Da bei der

Schlüsselerstellung die Nachrichtenlänge keine Relevanz hat, wird diesbezüglich keine Auswertung durchgeführt.

Durch die Auswertung der Messreihen kann nicht eindeutig bestimmt werden, welche Nachrichtenlänge generell mehr oder weniger CPU-Zyklen für die Ausführung benötigt Daher werden in den folgenden Abschnitten und Tabellen die Auswertungen der Messreihen unter Verwendung einer Nachrichtenlänge von 128 Byte und bei den klassischen Algorithmen auf die Hashfunktion sha512 beschränkt, die laut Preston *et al.* [74] auch die quantensicherste ist.

5.4.2.2. Auswertung der CPU-Zyklen bei der Schlüsselerstellung

Die folgenden Aussagen werden auf Basis der Mittelwerte getroffen. In Tabelle 5.5 werden alle CPU-Zyklen dargestellt, die für die Schlüsselerstellung unter Verwendung der jeweiligen Algorithmen gemessen wurden. Aus Tabelle 5.5 geht hervor, dass Dilithium3 im Vergleich zu den anderen Algorithmen am schnellsten bei der Erstellung von Schlüsselpaaren ist. secp256r1 ist der schnellste klassische Algorithmus, der ca. 12 Mal mehr CPU-Zyklen benötigt als Dilithium3. rsa4096 benötigt ca. 14.147 Mal und SPHINCS+-SHAKE256-192s-robust benötigt ca. 1.974 Mal mehr CPU-Zyklen als Dilithium3 und sind somit jeweils die langsamsten Algorithmen ihrer Klasse.

Generell benötigen die speicherplatzoptimierten Parametersets von SPHINCS+ ca. 56 bis 59 Mal mehr Zyklen als die geschwindigkeitsoptimierten Parametersets für Level 1 und 3. Für Level 5 benötigen die speicherplatzoptimierten Parametersets von SPHINCS+ ca. 15 Mal mehr CPU-Zyklen als die geschwindigkeitsoptimierten Parametersets. Die robust-Varianten sind ca. 1,8 Mal langsamer als die simple-Varianten für SHA256 Level 1 und 3 und Level 5 SHAKE256. Für Level 5 SHA256 sind die robust-Varianten ca. 3,4 Mal langsamer als die simple-Varianten. Somit ist die simple-Variante doppelt bis dreimal so schnell wie die robust-Variante.

Die Schlüsselerstellung benötigt bei Falcon-512 bzw. Falcon-1024 76 respektive 219 Mal länger als bei Dilithium3.

Bei 19 von 31 Algorithmen liegen die minimalen Werte im Rahmen der Standardabweichung. Die maximalen Werte liegen nicht im Rahmen der Standardabweichung.

5.4.2.3. Auswertung der CPU-Zyklen bei der Signaturerstellung

Die folgenden Aussagen werden auf Basis der Mittelwerte getroffen. In Tabelle 5.6 werden alle CPU-Zyklen dargestellt, die für die Signaturerstellung unter Verwendung der jeweiligen Algorithmen gemessen wurden. Aus Tabelle 5.6 geht hervor, dass Dilithium3 im Vergleich zu den anderen Algorithmen am schnellsten bei der Erstellung von Signaturen ist, gefolgt von Dilithium5 und Falcon-512 sowie Falcon-1024. Der Algorithmus SPHINCS+-SHAKE256-192s-robust benötigt ca. 9.732 Mal mehr CPU-Zyklen als Dilithium3 und ist somit am langsamsten. Bei den klassischen Algorithmen ist secp256r1 am schnellsten, aber

L	Algorithmus			Mittelwert (Cycles)	Median (Cycles)	Standard-abweichung (Cycles)	Minimaler Wert (Cycles)	Maximaler Wert (Cycles)
-			rsa2048	397.112.059	336.380.018	254.907.473	40.859.256	1.971.242.734
			rsa4096	4.254.680.764	3.603.927.232	2.908.547.659	212.638.835	24.512.202.849
			secp256r1	3.534.409	3.299.562	639.972	3.146.817	13.037.075
1			Falcon-512	22.759.020	21.045.430	5.907.319	16.203.509	61.710.902
	SPHINCS+	SHA256	128f-robust	3.262.297	2.975.035	716.795	2.787.383	12.385.411
			128f-simple	1.781.790	1.624.763	403.366	1.509.325	7.940.190
			128s-robust	184.296.598	182.899.016	9.503.367	173.130.667	332.352.046
			128s-simple	100.174.991	99.538.284	3.590.260	93.910.907	175.697.993
		SHAKE256	128f-robust	7.428.818	6.950.860	1.741.826	5.710.211	23.153.880
			128f-simple	4.021.943	3.693.146	1.003.958	3.317.588	18.352.276
			128s-robust	419.178.911	414.458.092	27.727.627	371.028.338	607.514.020
			128s-simple	230.808.864	227.203.906	19.495.229	204.518.917	396.689.864
3			Dilithium3	300.743	251.088	139.997	204.018	1.776.677
	SPHINCS+	SHA256	192f-robust	4.535.912	4.192.117	1.127.443	4.040.212	19.946.493
			192f-simple	2.579.888	2.384.791	542.706	2.238.848	9.528.491
			192s-robust	267.825.728	264.051.200	13.628.456	257.057.724	430.096.141
			192s-simple	147.357.530	144.962.498	7.576.394	140.684.807	285.001.542
		SHAKE256	192f-robust	10.266.427	9.845.368	1.503.168	8.347.741	30.784.720
			192f-simple	5.817.466	5.304.622	1.468.967	4.868.148	19.089.640
			192s-robust	593.878.619	588.303.072	30.480.024	545.742.338	844.180.308
			192s-simple	342.672.174	336.955.478	27.017.275	303.801.712	531.540.618
5			Dilithium5	410.747	354.211	151.422	294.796	2.607.259
			Falcon-1024	65.758.137	60.339.460	16.354.462	50.105.808	227.175.155
	SPHINCS+	SHA256	256f-robust	21.349.798	21.098.042	1.942.411	18.948.583	71.254.443
			256f-simple	6.282.131	5.698.364	1.253.660	5.457.794	26.100.961
			256s-robust	322.962.530	317.633.744	21.432.275	305.937.196	493.955.930
			256s-simple	93.143.807	91.929.185	5.986.577	86.583.979	203.974.574
		SHAKE256	256f-robust	26.395.655	25.593.006	3.615.928	22.369.860	83.988.066
			256f-simple	14.916.712	14.466.245	2.057.918	12.727.949	41.998.512
			256s-robust	397.752.949	391.848.819	27.015.170	364.096.385	642.789.455
			256s-simple	230.422.921	226.330.976	20.301.093	202.514.668	401.485.873

Tabelle 5.5.: Auswertung der CPU-Zyklen der Funktion `gen_keypair` von Algorithmen der Bibliotheken `python-mbedtls` und `liboqs-python`. L steht für Level, wobei „-“ für kein Level und somit die klassischen Algorithmen steht.

ist ca. 7 Mal langsamer als Dilithium3. rsa4096 ist der langsamte klassische Algorithmus, der ca. 14 Mal mehr CPU-Zyklen benötigt als secp256r1.

Generell benötigen die speicherplatzoptimierten Parametersets von SPHINCS+ ca. 16 bis 19 Mal mehr Zyklen als die geschwindigkeitsoptimierten Parametersets für Level 1 und 3. Für Level 5 benötigen die speicherplatzoptimierten Parametersets von SPHINCS+ ca. 8 bis 9 Mal CPU-Zyklen als die geschwindigkeitsoptimierten Parametersets. Die robust-Varianten sind ca. 1,7 bis 1,9 Mal langsamer als die simple-Varianten für SHA256 Level 1 und 3 und Level 5 SHAKE256. Für Level 5 SHA256 sind die robust-Varianten ca. 3,2 Mal langsamer als die simple-Varianten. Somit ist die simple-Variante zwei bis dreimal so schnell wie die robust-Variante.

L	Algorithmus			Mittelwert (Cycles)	Median (Cycles)	Standard-abweichung (Cycles)	Minimaler Wert (Cycles)	Maximaler Wert (Cycles)
-			rsa2048	11.054.261	10.722.024	1.262.677	10.379.275	35.728.107
			rsa4096	54.454.846	54.049.258	2.211.346	51.058.228	111.463.635
			secp256r1	3.812.381	3.559.204	662.553	3.373.693	12.402.504
1			Falcon-512	923.573	832.906	219.448	762.288	4.535.037
	SPHINCS+	SHA256	128f-robust	82.119.065	79.606.270	7.294.536	75.971.216	189.787.193
			128f-simple	43.789.247	43.105.640	4.381.893	39.899.754	115.150.594
			128s-robust	1.348.971.277	1.333.158.288	43.570.216	1.291.317.075	2.303.945.216
			128s-simple	718.696.223	716.276.206	13.709.138	707.833.195	1.284.484.789
		SHAKE256	128f-robust	179.451.457	175.070.128	17.629.180	157.457.453	318.189.666
			128f-simple	103.962.746	101.816.930	8.562.325	92.854.465	199.741.175
			128s-robust	3.048.306.126	3.048.752.912	95.202.676	2.710.401.307	3.879.631.533
			128s-simple	1.673.651.683	1.673.429.308	60.388.393	1.516.060.940	2.044.074.603
3			Dilithium3	534.011	441.538	317.155	229.369	3.116.232
	SPHINCS+	SHA256	192f-robust	132.396.750	129.221.969	10.895.801	123.616.517	261.736.964
			192f-simple	73.984.259	71.846.222	7.053.867	68.692.023	165.714.359
			192s-robust	2.492.905.109	2.488.957.632	32.880.047	2.454.358.567	2.852.179.570
			192s-simple	1.393.320.032	1.390.218.108	23.781.124	1.367.745.219	1.688.557.699
		SHAKE256	192f-robust	275.719.036	269.639.836	24.177.370	242.681.128	438.304.750
			192f-simple	161.507.971	157.435.028	14.873.700	142.112.926	288.680.247
			192s-robust	5.197.246.435	5.212.681.626	122.221.235	4.690.116.051	5.823.069.318
			192s-simple	3.076.841.312	3.074.878.266	76.031.847	2.825.506.357	3.699.227.893
5			Dilithium5	643.154	545.316	337.723	332.294	5.463.191
			Falcon-1024	1.800.617	1.643.014	405.184	1.533.495	7.036.597
	SPHINCS+	SHA256	256f-robust	421.200.951	414.557.173	27.133.107	399.228.353	665.385.587
			256f-simple	131.994.956	129.631.940	7.718.424	124.098.899	250.521.700
			256s-robust	3.612.937.638	3.623.840.965	59.015.706	3.524.842.031	3.982.222.365
			256s-simple	1.114.506.228	1.110.573.992	28.316.220	1.084.621.876	1.336.010.237
		SHAKE256	256f-robust	527.580.182	518.345.416	37.848.266	473.263.934	898.741.850
			256f-simple	301.455.891	296.718.966	21.552.488	271.264.172	483.238.917
			256s-robust	4.467.468.887	4.477.511.515	94.995.020	4.019.125.716	5.329.922.520
			256s-simple	2.653.550.216	2.649.342.503	66.017.279	2.458.736.956	3.173.449.961

Tabelle 5.6.: Auswertung der CPU-Zyklen der Funktion `sign` von Algorithmen der Bibliotheken `python-mbedtls` und `liboqs-python`.

Bei 15 von 31 Algorithmen liegen die minimalen Werte im Rahmen der Standardabweichung. Die maximalen Werte liegen nicht im Rahmen der Standardabweichung.

5.4.2.4. Auswertung der CPU-Zyklen bei der Signaturverifizierung

Die folgenden Aussagen werden auf Basis der Mittelwerte getroffen. In Tabelle 5.7 werden alle CPU-Zyklen dargestellt, die für die Signaturverifizierung unter Verwendung der jeweiligen Algorithmen gemessen wurden. Aus Tabelle 5.7 geht hervor, dass Dilithium3 im Vergleich zu den anderen Algorithmen am schnellsten bei der Verifizierung von Signaturen ist, gefolgt von Falcon-512, Dilithium5 und Falcon-1024. Der Algorithmus SPHINCS+-

L	Algorithmus			Mittelwert (Cycles)	Median (Cycles)	Standard-abweichung (Cycles)	Minimaler Wert (Cycles)	Maximaler Wert (Cycles)
-			rsa2048	279.735	265.846	64.273	248.759	1.418.671
			rsa4096	848.945	791.302	156.943	728.996	3.333.437
			secp256r1	7.214.049	6.869.426	962.627	6.560.369	19.428.410
1			Falcon-512	244.521	217.842	97.969	163.249	2.086.513
	SPHINCS+	SHA256	128f-robust	13.355.154	12.536.122	2.229.086	11.210.792	48.031.723
			128f-simple	6.902.159	6.509.127	1.236.570	5.804.506	31.901.492
			128s-robust	4.351.602	4.182.812	864.039	3.655.444	14.256.128
			128s-simple	2.236.852	2.204.986	299.844	1.903.472	9.024.593
		SHAKE256	128f-robust	21.311.036	20.377.314	3.779.205	16.180.300	67.988.151
			128f-simple	11.613.421	11.075.992	1.931.198	8.693.822	41.295.332
			128s-robust	7.208.798	7.132.662	757.454	5.487.211	20.582.221
			128s-simple	3.807.469	3.739.587	553.153	2.775.011	14.004.953
3			Dilithium3	208.293	180.862	67.509	171.684	1.419.565
	SPHINCS+	SHA256	192f-robust	19.496.514	18.455.271	3.354.541	17.244.627	66.326.839
			192f-simple	10.314.519	9.742.314	1.744.747	9.009.309	42.252.502
			192s-robust	6.591.655	6.538.390	469.929	5.907.429	16.737.740
			192s-simple	3.415.077	3.368.828	381.200	3.018.320	12.758.502
		SHAKE256	192f-robust	31.737.901	31.063.218	5.310.314	23.968.678	101.618.692
			192f-simple	16.703.478	15.929.610	3.150.544	12.424.502	54.514.898
			192s-robust	10.249.724	10.232.792	886.276	8.120.401	32.141.341
			192s-simple	5.373.030	5.344.820	496.621	4.212.093	17.586.880
5			Dilithium5	305.690	270.706	80.582	256.715	1.469.787
			Falcon-1024	406.840	394.215	122.202	300.364	2.047.782
	SPHINCS+	SHA256	256f-robust	24.546.816	24.043.316	2.609.326	22.456.307	85.728.414
			256f-simple	10.384.222	9.626.006	1.738.142	8.895.646	31.601.691
			256s-robust	12.322.329	12.256.458	788.708	11.229.059	34.769.343
			256s-simple	4.875.969	4.833.171	406.233	4.345.481	13.184.080
		SHAKE256	256f-robust	30.334.118	29.145.753	4.463.508	24.645.891	91.644.410
			256f-simple	16.137.057	15.816.140	2.119.780	12.729.461	54.727.696
			256s-robust	14.801.720	14.823.982	1.213.167	11.932.068	48.007.891
			256s-simple	7.853.349	7.800.105	868.717	6.202.935	27.020.467

Tabelle 5.7.: Auswertung der CPU-Zyklen der Funktion `verify` von Algorithmen der Bibliotheken `python-mbedtls` und `liboqs-python`.

SHAKE256-192f-robust benötigt ca. 152 Mal mehr CPU-Zyklen als Dilithium3 und ist somit am langsamsten. Bei den klassischen Algorithmen benötigt secp256r1 ca. 26 Mal mehr CPU-Zyklen als rsa2048. rsa2048 benötigt jedoch nur ca. 1,34 Mal mehr CPU-Zyklen als Dilithium3.

Bei der Verifizierung benötigen die speicherplatzoptimierten Parametersets weniger CPU-Zyklen als bei den geschwindigkeitsoptimierten Parametersets. Generell benötigen die geschwindigkeitsoptimierten Parametersets von SPHINCS+ bei der Verifizierung ca. 2,9 bis

3,1 Mal mehr Zyklen als die speicherplatzoptimierten Parametersets für SHA256 Level 1 und 3. Für Level 5 benötigen die geschwindigkeitsoptimierten Parametersets von SPHINCS+ ca. 2,0 bis 2,1 Mal mehr CPU-Zyklen als die speicherplatzoptimierten Parametersets. Die robust-Varianten sind ca. 1,8 bis 1,9 Mal langsamer als die simple-Varianten für SHA256 Level 1 und 3 und Level 5 SHAKE256. Für Level 5 SHA256 sind die robust-Varianten ca. 2,4 bis 2,5 Mal langsamer als die simple-Varianten. Somit ist die simple-Variante zwei bis zweieinhalb Mal so schnell wie die robust-Variante.

Bei 17 von 31 Algorithmen liegen die minimalen Werte im Rahmen der Standardabweichung. Die maximalen Werte liegen nicht im Rahmen der Standardabweichung.

5.4.2.5. Vergleich der Werte der Auswertungen

Abweichungen		Faktor (gen_keypair)	Faktor (sign)	Faktor (verify)
Mittelwert um Faktor höher als der Median	Median	1,031	1,016	1,027
	Minimaler Wert	1,006	0,997	0,998
	Maximaler Wert	1,198	1,209	1,152
	Mittelwert	1,054	1,031	1,040
Median um Faktor höher als die Min-Werte	Median	1,099	1,084	1,170
	Minimaler Wert	1,027	1,012	1,053
	Maximaler Wert	1,299	1,925	1,348
	Mittelwert	1,111	1,121	1,187
Mittelwert um Faktor höher als die Min-Werte	Median	1,144	1,101	1,222
	Minimaler Wert	1,042	1,015	1,093
	Maximaler Wert	1,474	2,328	1,498
	Mittelwert	1,173	1,165	1,234
Max-Werte um Faktor höher als der Median	Median	3,030	1,731	3,462
	Minimaler Wert	1,435	1,099	2,560
	Maximaler Wert	7,361	10,018	9,578
	Mittelwert	3,168	2,347	3,968
Max-Werte um Faktor höher als der Mittelwert	Median	2,907	1,706	3,397
	Minimaler Wert	1,421	1,102	2,539
	Maximaler Wert	6,348	8,494	8,533
	Mittelwert	2,948	2,197	3,779

Tabelle 5.8.: Abweichungen von minimalen und maximale Werten von den jeweiligen Mittelwerten und Medianen der PQC-Algorithmen bei den Funktionen.

In Tabelle 5.8 werden Vergleiche zwischen den verschiedenen Auswertungsergebnissen der Messreihen unter Verwendung der unterschiedlichen Algorithmen dargestellt. Es wird verglichen um wie viel Prozent:

- der Mittelwert höher ist als der Median,
- der Median bzw. Mittelwert jeweils höher ist als die minimalen Werte und

- die maximalen Werte jeweils höher sind als deren Median bzw. Mittelwert.

Es werden nur die Abweichungen betrachtet, die bei den PQC-Algorithmen bestimmt wurden.

Die Mittelwerte sind bis zu ca. 20,9 Prozent höher als die Mediane, durchschnittlich sind sie bis zu ca. 5,4 Prozent höher.

Minimale Werte. In Tabelle 5.8 ist aufgeführt, dass die Mediane und Mittelwerte bei der Schlüsselerstellung und Signaturverifizierung um bis zu 50 Prozent höher sind als die minimalen Werte. Bei der Signaturerstellung liegen die Abweichungen bei bis zu 232,8 Prozent. Im Durchschnitt liegen die Abweichungen bei bis maximal 23,4 Prozent.

Maximale Werte. In Tabelle 5.8 ist aufgeführt, dass die maximalen Werte bei allen drei Funktionen zwischen 634,8 und 1001,8 Prozent höher sind als die jeweiligen Mediane und Mittelwerte. Durchschnittlich liegen die Abweichungen zwischen 234,7 bzw. 396,8 Prozent.

In den Tabellen zu den CPU-Zyklen der Funktionen werden die absoluten maximalen Werte für alle Algorithmen dargestellt. Bei den Messwerten zu den `sign`-Funktionen (vgl. Tabelle 5.6) der Dilithium- und Falcon-Algorithmen fällt auf, dass die maximalen Werte bis zu 10 Mal höher sind als die Mediane und Mittelwerte. Die maximalen Messwerte bei allen `verify`-Funktionen (vgl. Tabelle 5.7) liegen mindestens 2,5 Mal höher als die Mittelwerte.

Algorithmen. Bei RSA dauert die Schlüsselerstellung länger als das Signieren, wobei das Signieren länger dauert als das Verifizieren (vgl. Tabellen 5.5, 5.6 und 5.7). Bei secp256r1 dauern `gen_keypair` und `sign` in etwa gleich lang, wobei `sign` etwas länger benötigt. `verify` dauert fast doppelt so lange wie `gen_keypair` und `sign`.

rsa2048 ist prinzipiell schneller als rsa4096. secp256r1 ist schneller bei der Schlüsselerstellung und beim Signieren als rsa2048 und rsa4096. Im Gegensatz dazu sind rsa2048 und rsa4096 beim Verifizieren schneller als secp256r1.

Auch bei den PQC-Algorithmen variiert, welche der Funktionen weniger CPU-Zyklen benötigt. Am Beispiel von Dilithium3 kann festgestellt werden, dass die Signaturerstellung länger dauert als die Schlüsselerstellung und das zum Verifizieren einer Signatur am wenigsten CPU-Zyklen benötigt werden (vgl. Tabellen 5.5, 5.6 und 5.7). Für Falcon-512 dauert die Schlüsselerstellung ca. 25 Mal länger als die Signaturerstellung und ca. 93 Mal länger als die Signaturverifizerung. Unter der Verwendung von Falcon-1024 dauert die Schlüsselerstellung ca. 36 Mal länger als die Signaturerstellung und ca. 161 Mal länger als die Signaturverifizerung. Beim Parameterset SPHINCS+-SHA256-128f-robust benötigt die Signaturerstellung mehr CPU-Zyklen als die Verifizierung und die Verifizierung mehr CPU-Zyklen als die Schlüsselerstellung. Beim Parameterset SPHINCS+-SHA256-128s-robust benötigt die Signaturerstellung mehr CPU-Zyklen als die Schlüsselerstellung und diese wiederum mehr CPU-Zyklen als die Schlüsselverifizierung.

Die geschwindigkeitsoptimierten SPHINCS+-SHA256-Parametersets benötigen zur Schlüsselerstellung ca. 1/8 der Zeit, die Falcon-512 benötigt. Die speicherplatzoptimierten

SPHINCS+-SHA256-Parametersets benötigen zur Schlüsselerstellung 8 Mal so lange wie Falcon-512. Die schnellsten Algorithmen zur Schlüsselerstellung sind Dilithium3 und Dilithium5 (vgl. Tabelle 5.5). Die schnellsten Algorithmen zur Signaturerstellung sind Dilithium3 und Dilithium5, gefolgt von Falcon-512 (vgl. Tabelle 5.6). Die schnellsten Algorithmen zur Signaturverifizierung sind Dilithium3, gefolgt von Falcon-512, Dilithium5 und Falcon-1024 (vgl. Tabelle 5.7). Die SPHINCS+-Parametersets benötigen für die `gen_keypair`- und `sign`-Funktionen bis zu 10.000 Mal mehr CPU-Zyklen und ca. 100 Mal mehr CPU-Zyklen für das Verifizieren. Falcon benötigt weniger CPU-Zyklen zum Signieren und Verifizieren als die SPHINCS+-SHA256-Parametersets. Bei der Schlüsselerstellung sind die geschwindigkeitsoptimierten Parametersets von SPHINCS+-SHA256 jedoch schneller als die Falcon-Parametersets.

Mittelwerte und Mediane sind bei allen Parametersets sehr ähnlich, werden jedoch beide in Tabelle 5.8 aufgeführt. Es fällt auf, dass die Mediane prinzipiell niedriger sind als die Mittelwerte, was zeigt, dass es einige Ausreißer nach oben gibt, jedoch der Hauptanteil der Messwerte innerhalb der Standardabweichung vom Median bzw. Mittelwert entfernt liegen muss.

SHAKE256 in der geschwindigkeitsoptimierten Version benötigt fast doppelt so viele CPU-Zyklen wie SHA256. Dies spiegelt sich in allen Berechnungen für Mittelwerte, Mediane, Standardabweichungen sowie Minima und Maxima wider. Es fällt auf, dass die CPU-Zyklen sehr viel höher sind als bei allen anderen in diesem Level dargestellten Algorithmen. Dies wurde durch die Designer schon bekannt gegeben, da die speicherplatzoptimierten Parametersets mehr Zeit benötigten, um die Signaturgröße kleiner zu halten [67].

5.4.2.6. Vergleich klassischer und quantensicherer Algorithmen

Dilithium3 benötigt bei den drei Funktionen am wenigsten CPU-Zyklen. Weitere Algorithmen wie Dilithium5 und Falcon-512 sowie Falcon-1024 sind bei der Signaturerstellung schneller als die betrachteten klassischen Algorithmen. Mithilfe der Tabellen der Auswertung zu den Funktionen in Abschnitt 5.4.2 werden folgende Vergleiche festgestellt. Bei der Schlüsselerstellung ist Dilithium3 ca. 11,8 Mal schneller als secp256r1, sowie ca. 1.320 Mal schneller als rsa2048 und ca. 14.057 Mal schneller als rsa4096. Bei der Signaturerstellung ist Dilithium3 ca. 7,1 Mal schneller als secp256r1, sowie ca. 20,7 Mal schneller als rsa2048 und ca. 102,8 Mal schneller als rsa4096. Bei der Signaturverifizierung ist Dilithium3 ca. 1,3 Mal schneller als rsa2048, sowie ca. 4,2 Mal schneller als rsa4096 und ca. 34,6 Mal schneller als secp256r1. SPHINCS+ ist bei der Signaturerstellung langsamer als die meisten klassischen und weitere quantensichere Algorithmen - wie Falcon und Dilithium5.

5.5. Messungen der CPU-Zyklen der Szenarien

In diesem Abschnitt wird die Durchführung der Messungen der CPU-Zyklen der Szenarien des Watchdog-Timer-Protokolls (Abschnitt 5.5.1) sowie die Auswertung der Messreihen (Abschnitt 5.5.2) beschrieben.

5.5.1. Durchführung der Messreihen

Da sich bei der Auswertung der quantensicheren Funktionen in Abschnitt 5.4.2.1 herausgestellt hat, dass die Unterschiede der Messergebnisse bei der Verwendung von drei verschiedenen Nachrichtenlängen (64 Byte, 96 Byte, 128 Byte) sehr ähnlich sind und nicht herausgelesen werden kann, dass die kürzere Nachricht auch weniger CPU-Zyklen benötigt, wird bei den Messreihen der Szenarien die gesamte Nachricht als Bytestring mithilfe der `sign`-Funktion gehasht und signiert. Die klassischen Szenario-Messungen verwenden ausschließlich die Hashfunktion `sha512` in den Funktionen `sign` und `verify`. Bei den quantensicheren Szenario-Messungen wird die bereits in der Signaturfunktion implementierte Hashfunktion verwendet, weshalb keine zu verwendende Hashfunktion angegeben werden muss.

Alle klassischen und quantensicheren Algorithmen, die in Tabelle 5.3 aufgeführt sind, wurden auch bei den Messungen der Szenarien verwendet.

Die Messpunkte für das Starten und Beenden des `hwcounter` befinden sich für die Szenarien 1, 2, 6, 7 und 8 im `TopLevelActor` (vgl. Abschnitt 5.2). Die Messung wird direkt vor dem Senden der ersten Nachricht (`"boot"`) an das Device bzw. (`"update"`) an den Server gestartet. Das Beenden der Messung findet direkt vor dem Shutdown des Aktorsystems und somit vor dem Shutdown des Geräts statt. Die einzige Ausnahme ist Szenario 3 (Abschnitt 3.1.4.4). Szenario 3 hat den gleichen Startpunkt der Messung wie die anderen Szenarien und zwei Endpunkte, die nicht im `TopLevelActor` sind. Bei dem Szenario findet zunächst die Initialisierung des Timers mit dem Starten des Timers statt und zeitgleich wird in die Business-Logik gebootet. Der Sensor misst einen Wert und gibt diesen zum Speichern an den Server weiter. Die zwei Endpunkte von Szenario 3 befinden sich im `Timer` des Geräts, nachdem der Timer initialisiert und einmal auf Ablauf der Timerzeit geprüft wurde und im `Storage` des Servers, nachdem der erste Messwert des `Sensor`s gespeichert wurde.

5.5.2. Auswertung der CPU-Zyklen der Szenarien

Die Auswertung der CPU-Zyklen der unterschiedlichen Szenarien erfolgt einzeln, jeweils im Vergleich zwischen den Algorithmen der drei NIST-Level und den drei klassischen Algorithmen. Bei der Ausführung der Szenarien unter Verwendung klassischer Kryptografie wird als Parameter für `sign` und `verify` die Hashfunktion SHA512 verwendet. Die Spalte L steht für das NIST-Sicherheitslevel, wobei „-“ dafür steht, dass es sich um einen klassischen kryptografischen Algorithmus handelt.

Die gemachten Angaben in Faktoren und Prozent beziehen sich jeweils auf die Mediane, Mittelwerte sowie minimalen und maximalen Werte der im Folgenden dargestellten Tabellen. Bei den genannten Vergleichen werden ausschließlich die quantensicheren Algorithmen untereinander verglichen.

5.5.2.1. Szenario 1: Kein Bootticket vorhanden

L	Algorithmus			Mittelwert (Cycles)	Median (Cycles)	Standard-abweichung (Cycles)	Minimaler Wert (Cycles)	Maximaler Wert (Cycles)
-			rsa2048	394.484.714	393.171.808	14.223.670	358.717.534	485.569.820
			rsa4096	483.778.452	482.928.076	12.184.870	460.831.432	563.274.964
			secp256r1	390.948.970	390.366.410	10.383.238	359.630.663	414.239.221
1			Falcon-512	350.788.515	346.403.546	18.325.896	301.245.860	402.464.913
	SPHINCS+	SHA256	128f-robust	557.625.058	557.949.518	11.112.995	532.546.307	585.964.374
			128f-simple	498.751.115	476.262.010	62.159.048	430.610.714	835.703.529
			128s-robust	3.013.703.345	3.000.863.569	77.157.244	2.956.689.448	3.656.536.246
			128s-simple	1.813.213.671	1.815.025.205	26.902.886	1.760.847.530	1.879.175.987
		SHAKE256	128f-robust	721.081.836	720.893.135	15.128.479	680.537.719	760.938.450
			128f-simple	582.070.067	580.873.602	16.768.857	537.802.025	631.574.772
			128s-robust	6.224.568.916	6.215.330.182	62.816.624	6.105.888.847	6.474.762.248
			128s-simple	3.645.362.276	3.639.254.897	37.742.940	3.587.356.117	3.869.305.543
3			Dilithium3	373.273.327	375.217.842	15.183.089	334.792.835	400.571.326
	SPHINCS+	SHA256	192f-robust	661.044.757	661.027.334	13.810.836	635.199.193	704.295.963
			192f-simple	536.982.833	539.550.246	17.290.990	499.080.763	609.253.700
			192s-robust	5.368.101.227	5.359.608.184	53.033.019	5.290.688.671	5.601.548.261
			192s-simple	3.138.100.796	3.132.676.020	25.157.829	3.097.008.250	3.291.453.804
		SHAKE256	192f-robust	933.766.014	933.077.424	15.147.534	896.429.823	965.983.208
			192f-simple	700.663.824	697.464.692	17.673.850	658.912.230	789.669.145
			192s-robust	10.253.040.895	10.243.978.649	195.746.496	9.682.499.302	10.933.542.301
			192s-simple	6.224.028.864	6.215.423.838	99.353.601	6.019.581.697	6.491.932.155
5			Dilithium5	376.832.005	379.894.780	16.067.862	338.499.929	406.689.303
			Falcon-1024	384.132.925	384.824.193	14.277.140	341.860.983	419.036.772
	SPHINCS+	SHA256	256f-robust	1.217.538.621	1.215.599.092	16.291.279	1.168.144.655	1.256.904.652
			256f-simple	646.714.846	647.584.406	12.295.273	619.594.982	684.747.860
			256s-robust	7.465.776.966	7.463.779.103	19.655.846	7.421.643.799	7.525.670.660
			256s-simple	2.563.747.405	2.561.908.020	16.418.455	2.521.093.808	2.604.546.547
		SHAKE256	256f-robust	1.394.850.175	1.393.050.292	19.770.960	1.353.653.660	1.449.259.983
			256f-simple	980.629.443	978.907.192	15.859.073	942.743.782	1.025.070.200
			256s-robust	8.792.846.301	8.775.718.569	124.141.542	8.546.135.100	9.071.004.532
			256s-simple	5.304.966.154	5.303.075.106	43.229.195	5.209.399.734	5.442.825.998

Tabelle 5.9.: Auswertung der CPU-Zyklen des Szenario 1 unter Verwendung der Algorithmen der Bibliotheken `python-mbedtls` und `liboqs-python`.

In Tabelle 5.9 werden die Auswertungen der Messreihen zu Szenario 1 dargestellt. Bei Szenario 1 finden zwei Signaturerstellungen und eine Signaturverifizierung statt, sowie zwei Versände über den Kommunikationskanal zwischen Gerät und Server. Bei Szenario 1 benötigen die robust-Varianten mehr CPU-Zyklen als die simple-Varianten. Die geschwindigkeitsoptimierten Parametersets benötigen weniger CPU-Zyklen als die speicherplatzoptimierten Parametersets. Der schnellste Algorithmus ist Falcon-512. Der langsamste Algorithmus ist SPHINCS+-SHAKE256-192s-robust und benötigt 29,2 Mal mehr CPU-Zyklen als Falcon-512.

5.5.2.2. Szenario 2: Bootticket vorhanden, aber nicht valide

L	Algorithmus			Mittelwert (Cycles)	Median (Cycles)	Standard-abweichung (Cycles)	Minimaler Wert (Cycles)	Maximaler Wert (Cycles)
-			rsa2048	423.876.002	419.990.519	19.231.158	386.526.121	478.883.083
			rsa4096	521.904.984	521.668.176	21.127.784	483.445.337	570.605.429
			secp256r1	434.320.042	428.539.334	20.818.168	401.978.843	498.700.427
1			Falcon-512	415.512.968	413.542.327	13.703.094	390.012.605	468.262.489
	SPHINCS+	SHA256	128f-robust	610.356.336	606.611.758	23.522.355	566.558.665	763.286.414
			128f-simple	510.110.982	501.757.066	20.526.657	476.850.801	570.464.547
			128s-robust	3.054.307.206	3.052.394.908	26.049.102	3.003.778.185	3.127.986.037
			128s-simple	1.858.657.225	1.855.080.561	31.162.881	1.797.567.045	2.059.916.317
		SHAKE256	128f-robust	783.799.713	782.420.402	15.054.109	751.510.083	821.626.145
			128f-simple	636.317.964	635.935.280	19.504.803	592.343.648	683.926.461
			128s-robust	6.244.341.312	6.242.511.260	59.205.464	6.144.130.195	6.397.763.759
			128s-simple	3.660.556.094	3.649.950.812	43.566.629	3.587.382.395	3.892.267.731
3			Dilithium3	411.909.928	409.862.388	12.716.258	389.675.407	477.248.503
	SPHINCS+	SHA256	192f-robust	727.724.158	724.561.470	20.504.741	691.209.554	860.615.363
			192f-simple	589.607.369	587.365.307	21.631.113	558.150.389	707.116.892
			192s-robust	5.422.851.512	5.415.803.678	41.187.926	5.349.912.220	5.584.438.374
			192s-simple	3.176.643.220	3.174.101.496	19.970.284	3.137.385.157	3.235.557.744
		SHAKE256	192f-robust	1.027.112.035	1.028.380.625	17.386.430	984.494.475	1.073.594.958
			192f-simple	762.725.119	761.406.687	15.206.903	724.549.972	796.294.538
			192s-robust	10.660.230.369	10.658.169.589	208.834.926	9.995.467.535	11.550.528.403
			192s-simple	6.298.176.373	6.299.140.417	59.137.653	6.120.401.801	6.476.346.284
5			Dilithium5	415.407.734	412.929.215	14.963.162	388.233.194	470.357.992
			Falcon-1024	418.425.927	414.342.761	15.552.011	394.346.988	485.414.640
	SPHINCS+	SHA256	256f-robust	1.289.288.592	1.291.083.720	19.748.475	1.243.490.625	1.338.957.941
			256f-simple	698.313.905	699.238.522	16.793.291	657.048.308	749.149.624
			256s-robust	7.527.038.053	7.527.482.529	21.813.516	7.474.568.172	7.581.311.216
			256s-simple	2.609.390.022	2.608.149.807	19.843.319	2.561.587.668	2.662.150.118
		SHAKE256	256f-robust	1.508.976.019	1.506.919.592	36.235.172	1.461.706.058	1.786.670.312
			256f-simple	1.064.375.852	1.062.032.088	17.685.634	1.022.448.963	1.112.618.471
			256s-robust	8.920.885.187	8.909.257.228	116.320.617	8.670.451.189	9.306.579.700
			256s-simple	5.350.450.962	5.343.742.796	40.631.552	5.254.886.442	5.437.412.911

Tabelle 5.10.: Auswertung der CPU-Zyklen des Szenario 2 unter Verwendung der Algorithmen der Bibliotheken `python-mbedtls` und `liboqs-python`.

In Tabelle 5.10 werden die Auswertungen der Messreihen zu Szenario 2 dargestellt. Bei Szenario 2 finden zwei Signaturerstellungen und zwei Signaturverifizierungen statt, sowie zwei Versände über den Kommunikationskanal zwischen Gerät und Server.

Bei Szenario 2 benötigen die robust-Varianten mehr CPU-Zyklen als die simple-Varianten. Die geschwindigkeitsoptimierten Parametersets benötigen weniger CPU-Zyklen als die speicherplatzoptimierten Parametersets. Der schnellste Algorithmus ist Dilithium3. Der langsamste Algorithmus ist SPHINCS+-SHAKE256-192s-robust und benötigt 25,9 Mal mehr CPU-Zyklen als Dilithium3.

5.5.2.3. Szenario 3a: Bootticket vorhanden und valide (Initialisieren des Timers)

L	Algorithmus			Mittelwert (Cycles)	Median (Cycles)	Standard-abweichung (Cycles)	Minimaler Wert (Cycles)	Maximaler Wert (Cycles)
-			rsa2048	200.846.744	201.211.385	11.311.941	176.798.978	242.897.970
			rsa4096	203.637.681	205.839.602	10.489.618	181.676.792	222.904.879
			secp256r1	215.290.551	215.610.998	8.393.909	185.724.652	231.797.309
1			Falcon-512	202.232.972	203.152.764	9.753.096	179.015.352	221.565.958
	SPHINCS+	SHA256	128f-robust	220.132.240	220.206.352	8.412.996	190.691.895	244.366.197
			128f-simple	207.816.909	208.844.734	11.178.315	177.669.551	237.101.863
			128s-robust	208.796.196	209.127.848	9.563.582	181.812.331	235.551.355
			128s-simple	204.980.466	206.508.198	10.271.754	178.480.997	233.400.905
		SHAKE256	128f-robust	211.824.109	214.974.110	11.272.033	185.695.194	232.867.670
			128f-simple	212.574.849	214.393.673	8.613.906	184.374.067	225.458.523
			128s-robust	198.601.584	199.966.698	10.054.780	180.977.175	225.260.949
			128s-simple	195.278.796	193.552.816	10.320.580	178.518.649	223.152.512
3			Dilithium3	204.262.693	206.328.274	10.269.349	181.544.112	232.745.665
	SPHINCS+	SHA256	192f-robust	226.994.823	226.460.602	8.120.016	206.411.480	255.517.118
			192f-simple	213.971.263	213.552.873	7.307.259	190.719.412	233.632.955
			192s-robust	212.064.852	211.457.442	8.787.084	189.884.050	242.894.336
			192s-simple	205.419.850	207.194.566	8.850.010	179.339.533	219.282.236
		SHAKE256	192f-robust	235.452.490	235.552.312	6.681.693	215.287.347	251.528.382
			192f-simple	221.270.794	221.414.758	7.069.139	192.741.759	234.424.219
			192s-robust	213.971.483	215.540.802	8.077.698	188.882.418	233.930.525
			192s-simple	203.894.497	206.605.684	10.442.074	172.098.645	220.800.017
5			Dilithium5	202.515.861	204.807.903	11.334.895	177.228.507	237.747.227
			Falcon-1024	204.407.302	205.377.646	9.468.289	180.722.413	230.146.474
	SPHINCS+	SHA256	256f-robust	226.627.537	227.963.972	7.969.960	208.793.048	253.278.709
			256f-simple	212.065.352	213.256.538	9.171.884	186.173.808	228.518.389
			256s-robust	212.057.223	213.845.461	9.557.562	182.958.399	231.388.697
			256s-simple	204.027.641	207.048.501	10.473.644	177.705.408	221.744.607
		SHAKE256	256f-robust	234.834.108	235.631.999	6.819.931	214.795.871	250.283.877
			256f-simple	222.790.512	222.848.182	9.127.920	195.047.629	269.961.036
			256s-robust	217.276.066	218.187.556	7.715.425	187.606.914	231.605.652
			256s-simple	209.080.432	211.254.972	8.525.966	186.470.807	225.434.935

Tabelle 5.11.: Auswertung der CPU-Zyklen des Szenario 3a unter Verwendung der Algorithmen der Bibliotheken `python-mbedtls` und `liboqs-python`.

In Tabelle 5.11 ist die Auswertung zum ersten Messpunkt des dritten Szenarios aufgeführt. Bei Szenario 3a findet keine Signaturerstellung und kein Versand zwischen Gerät und Server statt, aber es wird eine Signaturverifizierung durchgeführt.

Bei Szenario 3a benötigen die robust-Varianten mehr CPU-Zyklen als die simple-Varianten, mit der Ausnahme SPHINCS+-SHAKE256-128f. Die speicherplatzoptimierten sind schneller als die geschwindigkeitsoptimierten Parametersets. SPHINCS+-SHAKE256-192f-robust

ist 1,206 Mal langsamer als SPHINCS+-SHAKE256-128s-simple und ist somit der langsamste Algorithmus.

5.5.2.4. Szenario 3b: Bootticket vorhanden und valide (Speichern des Sensor-Messwerts)

L	Algorithmus			Mittelwert (Cycles)	Median (Cycles)	Standard-abweichung (Cycles)	Minimaler Wert (Cycles)	Maximaler Wert (Cycles)
-			rsa2048	312.751.673	311.389.820	10.967.498	288.146.973	378.680.714
			rsa4096	357.167.860	357.076.716	13.380.293	330.266.900	392.490.401
			secp256r1	325.096.300	323.869.275	11.999.463	300.314.880	366.566.408
1			Falcon-512	301.710.280	301.254.499	9.886.519	271.831.531	332.833.956
	SPHINCS+	SHA256	128f-robust	414.373.662	413.902.114	10.569.368	388.305.129	444.881.352
			128f-simple	356.126.443	355.178.936	13.040.466	328.821.629	386.731.534
			128s-robust	1.629.890.915	1.625.216.566	21.160.295	1.595.342.706	1.704.578.241
			128s-simple	1.027.779.676	1.024.526.178	18.033.131	993.913.760	1.106.986.138
		SHAKE256	128f-robust	497.022.823	496.146.452	13.359.526	474.194.309	525.545.996
			128f-simple	421.164.297	421.527.924	8.266.842	400.462.215	443.687.632
			128s-robust	3.222.066.965	3.221.562.697	71.359.969	3.060.836.269	3.379.615.714
			128s-simple	1.919.023.781	1.914.707.335	36.361.566	1.822.806.419	2.007.352.535
3			Dilithium3	305.174.678	304.192.380	9.937.172	287.300.029	358.812.623
	SPHINCS+	SHA256	192f-robust	479.784.114	478.518.588	13.835.664	457.145.610	515.692.862
			192f-simple	398.643.668	397.766.751	10.118.602	383.739.929	432.195.207
			192s-robust	2.812.186.714	2.806.190.196	25.664.328	2.768.964.254	2.913.405.983
			192s-simple	1.688.459.412	1.688.162.852	15.237.658	1.657.549.554	1.736.000.397
		SHAKE256	192f-robust	641.371.415	640.669.558	15.287.275	607.873.097	679.108.577
			192f-simple	496.657.540	497.722.791	14388609	463390705	530.796.708
			192s-robust	5.555.662.606	5.542.596.673	189.158.285	5.308.866.582	6.916.507.686
			192s-simple	3.292.449.450	3.286.514.694	50.045.224	3.168.550.656	3.395.092.034
5			Dilithium5	305.417.931	304.447.486	9.198.974	281.924.580	339.321.412
			Falcon-1024	305.982.414	305.841.951	8.833.668	279.174.447	332.654.542
	SPHINCS+	SHA256	256f-robust	765.287.163	764.440.972	14.794.748	735.160.645	810.930.854
			256f-simple	452.082.516	451.273.300	10.861.745	430.343.069	483.400.988
			256s-robust	3.869.760.479	3.867.511.234	18.656.953	3.837.135.611	3.936.128.334
			256s-simple	1.406.483.872	1.404.102.654	14.477.612	1.378.019.757	1.456.275.241
		SHAKE256	256f-robust	895.384.834	895.533.320	19.504.459	855.608.422	939.049.581
			256f-simple	648.226.224	648.150.928	17.419.590	612.730.197	765.653.440
			256s-robust	4.663.191.047	4.659.537.409	77.207.346	4.538.163.638	4.857.981.916
			256s-simple	2.810.639.987	2.806.461.458	38.722.809	2.738.340.072	3.044.339.456

Tabelle 5.12.: Auswertung der CPU-Zyklen des Szenario 3b unter Verwendung der Algorithmen der Bibliotheken `python-mbedtls` und `liboqs-python`.

In Tabelle 5.12 ist die Auswertung zum zweiten Messpunkt des dritten Szenarios aufgeführt. Bei Szenario 3b finden eine Signaturerstellung und zwei Signaturverifizierung statt, sowie ein Versand über den Kommunikationskanal zwischen Gerät und Server.

Bei Szenario 3b benötigen die robust-Varianten mehr CPU-Zyklen als die simple-Varianten.

Die geschwindigkeitsoptimierten Parametersets benötigen weniger CPU-Zyklen als die speicherplatzoptimierten Parametersets. Der schnellste Algorithmus ist Falcon-512. Der langsamste Algorithmus ist SPHINCS+-SHAKE256-192s-robust und benötigt 18,4 Mal mehr CPU-Zyklen als Falcon-512.

5.5.2.5. Szenario 6: Update vorhanden, aber nicht valide

L	Algorithmus			Mittelwert (Cycles)	Median (Cycles)	Standard-abweichung (Cycles)	Minimaler Wert (Cycles)	Maximaler Wert (Cycles)
-			rsa2048	424.943.746	417.907.452	21.536.111	392.228.553	541.321.740
			rsa4096	530.481.479	532.890.268	21.544.260	484.985.226	620.960.372
			secp256r1	429.732.696	423.271.040	20.292.260	397.235.795	479.196.566
1			Falcon-512	416.880.241	413.610.177	15.501.442	388.576.549	468.876.661
	SPHINCS+	SHA256	128f-robust	608.722.466	603.948.392	21.260.660	565.638.879	659.603.152
			128f-simple	518.160.131	514.222.404	21.688.062	479.069.139	568.237.577
			128s-robust	3.049.262.866	3.045.539.319	29.465.885	3.005.945.950	3.202.063.111
			128s-simple	1.863.916.546	1.862.418.246	25.184.387	1.816.653.858	1.934.974.396
		SHAKE256	128f-robust	778.280.507	777.360.144	15.938.441	743.086.736	812.827.921
			128f-simple	643.338.392	645.204.004	18.663.118	598.317.869	686.178.234
			128s-robust	6.220.180.318	6.219.419.660	71.537.553	5.990.174.006	6.442.953.242
			128s-simple	3.658.071.900	3.657.067.460	96.618.946	3.506.445.180	4.497.976.083
3			Dilithium3	417.481.764	414.208.041	15.136.213	388.067.715	463.118.095
	SPHINCS+	SHA256	192f-robust	729.198.122	72.9345.831	17.944.017	691.612.171	796.358.879
			192f-simple	585.541.650	583.280.005	17.171.169	547.385.336	652.079.156
			192s-robust	5.361.011.105	5.355.657.476	35.861.261	5.308.324.402	5.553.408.225
			192s-simple	3.175.688.702	3.173.729.392	23.401.842	3.136.061.822	3.336.637.871
		SHAKE256	192f-robust	1.016.425.811	1.016.581.972	17.452.014	975.804.126	1.064.343.399
			192f-simple	763.609.939	764.588.218	15.647.741	720.607.228	800.992.968
			192s-robust	10.620.739.822	10.601.435.043	216.601.378	9.962.857.100	11.142.350.950
			192s-simple	6.315.554.139	6.313.482.542	79.807.621	6.100.961.125	6.562.744.516
5			Dilithium5	416.704.806	412.454.823	16.044.929	393.958.636	465.548.015
			Falcon-1024	418.043.992	416.026.104	15.736.001	395.272.931	507.166.183
	SPHINCS+	SHA256	256f-robust	1.286.268.416	1.285.538.908	18.730.073	1.241.501.842	1.339.627.177
			256f-simple	703.676.122	702.323.984	15.221.823	665.797.014	747.613.470
			256s-robust	7.517.295.673	7.514.856.386	22.800.529	7.469.778.621	7.589.524.833
			256s-simple	2.607.352.803	2.607.315.214	15.363.452	2.574.734.475	2.659.682.549
		SHAKE256	256f-robust	1.485.980.364	1.484.471.663	25.810.970	1.426.169.261	1.561.758.640
			256f-simple	1.045.579.522	1.046.662.484	19.129.148	997.103.490	1.092.672.972
			256s-robust	8.958.414.327	8.922.764.463	158.630.746	8.631.427.896	9.672.782.205
			256s-simple	5.382.300.374	5.374.383.750	56.088.884	5.283.759.543	5.542.851.245

Tabelle 5.13.: Auswertung der CPU-Zyklen des Szenario 6 unter Verwendung der Algorithmen der Bibliotheken `python-mbedtls` und `liboqs-python`.

In Tabelle 5.13 werden die Auswertungen der Messreihen zu Szenario 6 dargestellt. Bei Szenario 6 finden zwei Signaturerstellungen und zwei Signaturverifizierung statt, sowie zwei Versände über den Kommunikationskanal zwischen Gerät und Server.

Bei Szenario 6 benötigen die robust-Varianten mehr CPU-Zyklen als die simple-Varianten.

Die geschwindigkeitsoptimierten Parametersets benötigen weniger CPU-Zyklen als die speicherplatzoptimierten Parametersets. Der schnellste Algorithmus ist Dilithium5. Der langsamste Algorithmus ist SPHINCS+-SHAKE256-192s-robust und benötigt 25,5 Mal mehr CPU-Zyklen als Dilithium5.

5.5.2.6. Szenario 7: Update vorhanden und valide

L	Algorithmus			Mittelwert (Cycles)	Median (Cycles)	Standard-abweichung (Cycles)	Minimaler Wert (Cycles)	Maximaler Wert (Cycles)
-			rsa2048	245.834.021	245.806.126	10.502.884	229.040.565	305.371.695
			rsa4096	250.386.022	248.933.024	11.159.389	229.092.741	303.044.305
			secp256r1	257.401.502	256.324.965	11.855.339	233.613.072	303.879.583
1			Falcon-512	247.288.915	246.107.226	10.503.076	222.313.404	306.491.224
	SPHINCS+	SHA256	128f-robust	261.531.190	258.698.501	12.534.257	233.726.723	303.746.368
			128f-simple	251.390.706	249.805.232	11.212.194	232.343.925	311.668.702
			128s-robust	246.707.561	245.483.068	8.567.663	224.480.137	278.028.403
			128s-simple	248.573.904	246.219.122	11.942.925	225.761.201	296.607.712
		SHAKE256	128f-robust	249.712.049	249.652.466	7.642.918	233.209.322	282.523.458
			128f-simple	253.977.690	253.709.696	7.216.403	239.626.255	275.936.876
			128s-robust	242.944.876	242.947.695	6.705.903	226.559.627	263.141.768
			128s-simple	241.745.760	242.211.389	7.474.318	218.528.211	257.479.790
3			Dilithium3	250.508.705	250.261.442	9.130.980	229.212.090	281.697.612
	SPHINCS+	SHA256	192f-robust	270.046.981	269.922.875	16.001.159	241.286.689	318.815.151
			192f-simple	257.124.837	254.952.135	13.463.407	229.482.256	317.411.200
			192s-robust	246.972.260	246.974.851	7.297.984	229.793.498	266.222.071
			192s-simple	245.783.009	245.771.741	8.486.590	222.827.829	281.178.178
		SHAKE256	192f-robust	282.627.796	285.415.531	16.239.277	249.047.626	319.985.023
			192f-simple	252.859.858	253.090.685	7.033.515	234.297.505	269.226.301
			192s-robust	253.420.091	253.607.567	7.548.619	234.162.854	285.720.406
			192s-simple	248.503.640	249.086.226	6.495.633	233.050.064	262.459.523
5			Dilithium5	249.215.903	249.128.884	8.310.261	228.832.625	274.694.938
			Falcon-1024	248.086.843	247.139.822	11.234.924	229.609.143	304.485.561
	SPHINCS+	SHA256	256f-robust	268.825.804	264.935.485	14.806.525	236.363.040	310.383.029
			256f-simple	250.513.060	250.383.772	8.370.045	232.682.141	286.517.158
			256s-robust	250.070.144	249.964.716	8.358.812	234.912.618	297.896.823
			256s-simple	246.517.641	246.339.576	5.826.280	233.630.210	260.587.219
		SHAKE256	256f-robust	284.841.112	288.945.999	15.844.397	248.016.787	312.232.012
			256f-simple	256.487.903	254.922.290	10.490.944	234.841.036	288.646.410
			256s-robust	249.769.090	249.994.336	7.414.518	235.890.529	276.506.645
			256s-simple	251.261.970	252.087.324	7.789.226	235.413.367	283.907.159

Tabelle 5.14.: Auswertung der CPU-Zyklen des Szenario 7 unter Verwendung der Algorithmen der Bibliotheken `python-mbedtls` und `liboqs-python`.

In Tabelle 5.14 werden die Auswertungen der Messreihen zu Szenario 7 dargestellt. Bei Szenario 7 findet keine Signaturerstellung, aber eine Signaturverifizierung statt, sowie kein

Versand über den Kommunikationskanal zwischen Gerät und Server.

Bei Szenario 7 benötigen mal die robust-Varianten mehr CPU-Zyklen als die simple-Varianten und mal vice versa. Die speicherplatzoptimierten Parametersets benötigen weniger CPU-Zyklen als die geschwindigkeitsoptimierten Parametersets, anders als bei den vorherigen Szenario-Messungen. Der langsamste Algorithmus ist SPHINCS+-SHAKE256-256f-robust und benötigt 1,178 Mal mehr CPU-Zyklen als der schnellste Algorithmus - SPHINCS+-SHAKE256-128s-simple.

5.5.2.7. Szenario 8: Server sendet Update

L	Algorithmus			Mittelwert (Cycles)	Median (Cycles)	Standard-abweichung (Cycles)	Minimaler Wert (Cycles)	Maximaler Wert (Cycles)
-			rsa2048	220.366.545	221.677.422	9.112.785	190.601.247	243.513.650
			rsa4096	255.517.650	251.727.276	13.716.682	235.551.036	297.857.643
			secp256r1	217.014.420	219.878.484	11.217.078	184.888.024	239.346.598
1			Falcon-512	213.988.489	215.019.184	11.127.462	182.910.764	235.562.603
	SPHINCS+	SHA256	128f-robust	290.949.867	290.693.414	12.923.828	258.687.649	316.716.412
			128f-simple	242.771.477	241.158.319	11.705.966	225.869.991	302.557.621
			128s-robust	1.509.866.142	1.510.125.630	20.489.981	1.461.451.210	1.605.500.477
			128s-simple	924.769.030	924.972.801	21.498.370	882.993.904	1.016.348.537
		SHAKE256	128f-robust	374.279.417	371.832.923	16.133.204	335.484.376	431.391.451
			128f-simple	310.012.947	311.000.306	11.088.849	288.656.819	336.492.561
			128s-robust	3.153.809.542	3.151.178.710	40.398.963	3.062.219.826	3.261.054.838
			128s-simple	1.847.468.108	1.846.294.031	23.770.182	1.799.253.913	1.906.098.369
3			Dilithium3	215.062.750	217.514.254	12.550.686	180.932.572	238.446.038
	SPHINCS+	SHA256	192f-robust	335.225.330	333.733.674	14.461.878	296.443.142	377.920.313
			192f-simple	282.660.599	283.089.332	14.664.392	251783465	311.231.101
			192s-robust	2.681.938.023	2.668.189.866	87.786.094	2.631.840.117	3.509.010.402
			192s-simple	1.584.365.568	1.584.379.669	16.641.770	1.544.301.400	1.631.660.148
		SHAKE256	192f-robust	480.419.714	480.375.406	13.942.804	451.601.817	523.933.225
			192f-simple	362.766.776	363.900.337	13.899.315	326.029.335	400.031.496
			192s-robust	5.180.177.769	5.235.080.601	148.557.113	4.796.842.716	5.387.420.731
			192s-simple	3.231.030.404	3.252.538.848	75.306.305	2.928.705.392	3.328.609.186
5			Dilithium5	215.166.051	217.205.916	12.767.888	183.826.466	250.766.426
			Falcon-1024	212.197.914	213.343.736	11.304.169	186.299.833	236.574.744
	SPHINCS+	SHA256	256f-robust	615.225.690	615.758.648	12.879.270	577.626.017	649.106.624
			256f-simple	333.487.062	331.909.932	11.978.281	310.222.123	371.643.465
			256s-robust	3.747.087.337	3.745.830.954	21.901.745	3.700.896.469	3.841.876.279
			256s-simple	1.297.514.162	1.298.954.906	15.392.575	1.254.412.842	1.339.520.768
		SHAKE256	256f-robust	688.940.184	688.142.218	17.497.033	644.493.552	734.784.741
			256f-simple	491.839.265	491.705.148	15.252.790	456.984.680	552.224.373
			256s-robust	4.514.881.119	4.523.810.738	89.150.486	4.274.734.602	4.733.481.057
			256s-simple	2.767.901.655	2.791.414.134	64.563.339	2.588.962.233	2.861.630.737

Tabelle 5.15.: Auswertung der CPU-Zyklen des Szenario 8 unter Verwendung der Algorithmen der Bibliotheken `python-mbedtls` und `liboqs-python`.

In Tabelle 5.15 werden die Auswertungen der Messreihen zu Szenario 8 dargestellt. Bei Szenario 8 findet eine Signaturerstellungen und keine Signaturverifizierung statt, sowie ein Versand über den Kommunikationskanal zwischen Gerät und Server.

Bei Szenario 8 benötigen die robust-Varianten mehr CPU-Zyklen als die simple-Varianten. Die geschwindigkeitsoptimierten Parametersets benötigen weniger CPU-Zyklen als die speicherplatzoptimierten Parametersets. Der schnellste Algorithmus ist Falcon-1024. Der langsamste Algorithmus ist SPHINCS+-SHAKE256-192s-robust und benötigt 24,4 Mal mehr CPU-Zyklen als Falcon-1024.

5.5.2.8. Vergleich der Werte der Auswertungen

Abweichungen		Szenario 1	Szenario 2	Szenario 3a	Szenario 3b	Szenario 6	Szenario 7	Szenario 8
Mittelwert um Faktor höher als der Median	Median	1,001	1,001	0,995	1,002	1,001	1,000	1,000
	Minimaler Wert	0,992	0,999	0,985	0,998	0,997	0,986	0,989
	Maximaler Wert	1,047	1,017	1,009	1,003	1,010	1,015	1,007
	Mittelwert	1,002	1,003	0,995	1,001	1,002	1,001	0,999
Median um Faktor höher als die Min-Werte	Median	1,041	1,040	1,144	1,047	1,045	1,084	1,073
	Minimaler Wert	1,006	1,007	1,084	1,008	1,006	1,054	1,012
	Maximaler Wert	1,150	1,074	1,201	1,108	1,078	1,165	1,202
	Mittelwert	1,052	1,039	1,139	1,048	1,043	1,090	1,084
Mittelwert um Faktor höher als die Min-Werte	Median	1,041	1,042	1,140	1,049	1,045	1,086	1,074
	Minimaler Wert	1,006	1,007	1,085	1,009	1,006	1,055	1,012
	Maximaler Wert	1,164	1,077	1,185	1,110	1,082	1,148	1,189
	Mittelwert	1,054	1,042	1,133	1,049	1,046	1,092	1,082
Max-Werte um Faktor höher als der Median	Median	1,053	1,058	1,092	1,064	1,051	1,129	1,090
	Minimaler Wert	1,008	1,007	1,052	1,018	1,010	1,054	1,023
	Maximaler Wert	1,755	1,258	1,211	1,248	1,230	1,248	1,315
	Mittelwert	1,090	1,087	1,102	1,078	1,075	1,138	1,092
Max-Werte um Faktor höher als der Mittelwert	Median	1,053	1,056	1,097	1,064	1,050	1,129	1,090
	Minimaler Wert	1,008	1,007	1,059	1,017	1,010	1,056	1,025
	Maximaler Wert	1,676	1,251	1,212	1,245	1,230	1,240	1,308
	Mittelwert	1,087	1,084	1,108	1,076	1,072	1,136	1,094

Tabelle 5.16.: Abweichungen von minimalen und maximale Werten von den jeweiligen Mittelwerten und Medianen der PQC-Algorithmen bei den Szenarien.

In Tabelle 5.16 werden Vergleiche zwischen den verschiedenen Auswertungsergebnissen der Szenarien unter Verwendung der unterschiedlichen Algorithmen dargestellt. Es wird verglichen um wie viel Prozent:

- der Mittelwert höher ist als der Median,
- der Median bzw. Mittelwert jeweils höher ist als die minimalen Werte und
- die maximalen Werte jeweils höher sind als deren Median bzw. Mittelwert.

Es werden nur die Abweichungen betrachtet, die bei den PQC-Algorithmen bestimmt wurden. Die Abweichungen werden im Abschnitt 5.16 diskutiert.

Die Mittelwerte und Mediane weichen bis zu 4,7 Prozent voneinander ab, durchschnittlich sind es ca. 0,5 Prozent.

Häufig ist SHAKE langsamer als SHA. Die Dilithium- und Falcon-Parametersets sind bei den Szenarien 1, 2, 3b, 6 und 8 am schnellsten.

Des Weiteren fällt auf, dass die Unterscheidung in die NIST-Sicherheitslevel 1, 3 und 5 nur bedingt Auswirkungen auf die Performance der Szenarien hat, da bei Szenario 6 und 8 jeweils ein Algorithmus des Level 5 am schnellsten ist.

Minimale Werte. In Tabelle 5.16 ist aufgeführt, um wie viel Prozent die Mediane und Mittelwerte im Vergleich zu den minimalen Werten höher sind. Die Mediane liegen bis zu 20,2 Prozent über den minimalen Werten, durchschnittlich bis zu 9,0 Prozent. Eine Ausnahme stellt Szenario 3a dar, bei dem der Median im Mittelwert um bis zu 13,9 Prozent höher liegt als die minimalen Werte. Die Mittelwerte liegen um bis zu 18,9 Prozent höher als minimalen Werte, durchschnittlich um bis zu 9,2 Prozent. Bei Szenario 3a liegt der Median im Durchschnitt bis zu 13,3 Prozent über den minimalen Werten.

Maximale Werte. In Tabelle 5.16 ist aufgeführt, um wie viel Prozent höher die maximalen Werte im Vergleich zu den jeweiligen Medianen und Mittelwerten sind. Die maximalen Werte liegen um bis zu 25,8 Prozent höher als die Mediane. Eine Ausnahme stellt Szenario 8 dar, bei dem die maximalen Werte um bis zu 31,5 Prozent über den Medianen liegen Bei Szenario 1 liegen die maximalen Werte um bis zu 75,5 Prozent über den Medianen. Die maximalen Werte liegen bis zu 30,8 Prozent über den Mittelwerten mit der Ausnahme von Szenario 1. Die maximalen Werte bei Szenario 1 liegen 67,6 Prozent über den Mittelwerten. Durchschnittlich liegen die maximalen Werte um bis zu 13,8 Prozent bzw. 13,6 Prozent höher als die Mediane bzw. Mittelwerte.

6. Diskussion

In diesem Kapitel wird die Umsetzung des Protokolls (Abschnitt 6.1) in Bezug auf die Anforderungen (Abschnitt 6.1.1) und die Implementierung des Protokolls (Abschnitt 6.1.2) diskutiert. Anschließend werden Diskussionen zu den Messungen (Abschnitt 6.2) in Bezug auf die getesteten Algorithmen (Abschnitt 6.2.1) sowie die Ergebnisse der Funktions- und Szenario-Messungen (Abschnitte 6.2.2 und 6.2.3) durchgeführt. In Abschnitt 6.3 wird evaluiert, auf welche Eigenschaften bei der Wahl des Algorithmus für das Protokoll bei der Anwendung im IoT-Bereich geachtet werden soll (Abschnitte 6.3.1 und 6.3.2) und welche Anwendungsfälle für das Watchdog-Timer-Protokoll möglich sind (Abschnitt 6.3.3). Außerdem wird die verwendte Methodik (Abschnitt 6.4) in Bezug auf die Implementierung (Abschnitt 6.4.1) und die Durchführung der Messreihen (Abschnitt 6.4.2) beurteilt. Parallel zu einigen Diskussionen werden Ideen für zukünftige Forschungsmöglichkeiten genannt, die im Abschnitt 7.4 zusammengefasst werden.

6.1. Diskussion der Umsetzung des Watchdog-Timer-Protokolls

In diesem Abschnitt wird die Umsetzung der Anforderungen (Abschnitt 6.1.1) und des Protokolls (Abschnitt 6.1.2) diskutiert und bewertet.

6.1.1. Umsetzung der Anforderungen

Die in Abschnitt 4.1 aufgeführten funktionalen und nicht-funktionalen Anforderungen wurden mit der Umsetzung des Konzepts als Proof-of-Concept-Implementierung umgesetzt. Die leichte Austauschbarkeit der klassischen durch quantensichere Algorithmen wurde durch die Parameter `crypto` und `variant` eingeführt. Schlüsselpaare sind quantensicher, wenn sie mit einem quantensicheren Algorithmus erstellt wurden. In diesem Zusammenhang ist außerdem das Thema *quantensichere Zertifikate* zu nennen, das jedoch in dieser Arbeit nicht betrachtet wird. Dies wäre ein weiteres Thema für zukünftige Forschungen.

Die Implementierung der Kommunikation zwischen einem Server und einem IoT-Gerät erfolgt komplett in Software mit dem Watchdog-Timer-Protokoll, welches in Kapitel 3 neu konzipiert und in Kapitel 4 implementiert wurde. Daher wird der Netzwerkverkehr nicht betrachtet. Das Protokoll wurde manuell auf seine Funktionsweise und die Abläufe nach den in Abschnitt 3.1.4 aufgeführten Szenarien überprüft. Es wurden alle in Abschnitt 3.1.4 entwickelten Szenarien und die Nachrichtentypen aus Abschnitt 3.1.5 in der Implementierung umgesetzt. Bei den Messungen wurden aufgrund von nebenläufigen Prozessen nur sechs

der acht Szenarien berücksichtigt (vgl. Abschnitt 5.5.1). Es wurden alle in Abschnitt 3.1.5 aufgeführten Schutzziele für die Nachrichtentypen durch entsprechende Implementierungen von kryptografischen Funktionen erreicht.

Mithilfe der Annahme, dass die Netzwerkverbindung sicher ist und verschlüsselt wird, wurde die Vertraulichkeit als gegeben angenommen. Zur Sicherung der Integrität und Authentizität wurden digitale Signaturverfahren eingesetzt, die Verfügbarkeit wurde durch den Einsatz eines Watchdog-Timers sichergestellt. Durch das Hashen und Signieren des Payloads kann sichergestellt werden, dass der Payload bei der Kommunikation über einen sicheren oder unsicheren Kanal nicht verändert wurde. Die Metadaten - abgesehen von der Signatur - können jedoch nicht auf ihre Integrität überprüft werden. Bei den klassischen Signaturalgorithmen wurden die Hashfunktionen SHA256, SHA384 oder SHA512 genutzt, wohingegen die quantensicheren Algorithmen SHA256, SHAKE128 und SHAKE256 nutzen. Bei der Verifizierung von `Request`s und `MeasuredData` wird angenommen, dass die Verifizierung der Signatur ausreicht, da keine Noncen vom Gerät erstellt werden können, wenn die Dateien vom Server aus gesendet, aber nicht wieder zur Bestätigung an diesen zurückgesendet werden. `Update`s können keine Noncen zur Verifizierung verwenden, da diese keine Rückmeldung vom Gerät erhalten, ob das Verifizieren des Updates erfolgreich war. Für diesen Fall gibt es die Versionsnummern, die allerdings für eine Anwendung in der realen Welt an die existierenden Standards angepasst werden müssen.

Auch die nicht-funktionalen Anforderungen wurden umgesetzt, da das Proof-of-Concept in Python mit dem Aktor-Framework Thespian implementiert wurde. Die Schlüsselverteilung wurde mit einem vorgeschalteten Skript umgesetzt und ist somit nicht im Protokoll enthalten. Außerdem wurde darauf geachtet, dass das Design des Watchdog-Timer-Protokolls größtenteils modular aufgebaut ist. Die Änderung der Aufgaben von Aktoren kann außerhalb des Aktorcodes in den Hilfsfunktionen erfolgen, sodass nicht direkt am Aktor-Code etwas geändert werden muss. Ein Beispiel wäre die Datenverarbeitung des Servers, der Sensordaten vom Gerät erhält. Die Funktion zur Datenverarbeitung kann angepasst werden. Aktoren, die zwischen zwei bestehenden Aktoren eingefügt werden müssten, können als Klassen im Aktor-Code ergänzt werden. Bei der Eingliederung eines Aktors zwischen zwei anderen Aktoren müssen jedoch diese drei Aktoren geändert werden, je nachdem von wem ein Aktor eine Nachricht erhält kann die auszuführenden Aufgabe variieren. Es bestehen daher noch Verbesserungsmöglichkeiten.

6.1.2. Umsetzung der Implementierung des Protokolls

Die Implementierung dient im Zusammenhang mit den Messskripten als Vergleichgrundlage mit der die unterschiedlichen Auswirkungen der klassischen und quantensicheren Algorithmen auf das Watchdog-Timer-Protokoll herausgearbeitet werden können. Das Protokoll wurde in drei verschiedenen Versionen umgesetzt. Die aktuell beigefügte Version ist die Proof-of-Concept-Implementierung mit der die einzelnen Szenario-Messungen durchgeführt werden können. Wie in Abschnitt 5.2 beschrieben, wurden Messpunkte eingefügt und die Implementierung dafür so weit angepasst, dass sie an zwei Stellen nicht weiterläuft, sondern das Aktorsystem beendet, weil die Messung beendet und eine neue Messung durchgeführt

wird. Soll das Protokoll wie in Kapitel 3 und 4 beschrieben funktionieren, müssen zwei Zeilen angepasst werden. Daher wurden die zwei weiteren Versionen der CD beigefügt. Eine Version enthält die zwei Änderungen zum Ablauf der Implementierung. Dadurch wird aus der Variante mit Messpunkten, die Variante die in der Konzeption (Kapitel 3) und in der Implementierung (Kapitel 4) umgesetzt wurde. Die andere Version enthält die vorherige Variante zum Messen der CPU-Zyklen der Szenarien mit zusätzlichem Hashen vor dem Signieren und dem Vergleichen des Hashes vor dem Verifizieren (Abschnitt 6.2.3.3). Die Einleitung zur Nutzung dieser Versionen befindet sich in Anhang A.2.

Bei der Umsetzung des Protokolls wurden teilweise Vereinfachungen vorgenommen, für die in realen Anwendungen Alternativen gefunden werden sollten. Zu den Vereinfachung zählen u.a.:

- In dieser Implementierung wird z.B. mit Integern als Versionsnummern für die Updates gearbeitet. Bei der Schlüsselerstellung wird die erste Versionssnummer als 0 in einer Textdatei gespeichert. Sind die Versionsnummern komplexer (z.B. 0.7.2) sollte über eine andere Art der Implementierung der `Update`-Verifizierung nachgedacht werden. Bei der Umsetzung einer realen Anwendung des Protokolls muss hier darauf geachtet werden, dass die realen Updates versendet, empfangen und installiert werden können.

- Aktuell kann nur jeweils nur ein `Update` und ein `Bootticket` in der `StagingArea` vorhanden sein kann. Wenn ein zweites `Update` bzw. `Bootticket` eintrifft bevor das vorherige installiert bzw. eingesetzt wurde, wird das alte mit dem neuen überschrieben. Daher sollte die Staging Area angepasst werden, sodass mehr als ein Update darin vorliegen kann und diese nacheinander installiert werden können.

- Der Sensor des Geräts wählt nur eine zufällige Zahl zwischen 4 und 14 als „Messwert“ aus und gibt diesen an den Server weiter. In einer realen Anwendung müsste das Protokoll überprüft werden und ggf. Schnittstellen zu einem tatsächlichen Sensor implementiert werden.

Das neu konzipierte Watchdog-Timer-Protokoll könnte als weiterer Forschungsschritt zusätzlich in Hardware implementiert werden. Ein Beispiel wäre der Prototyp von Lazarus [35], der mit quantensicheren Signaturalgorithmen ausgestattet werden könnte.

6.1.2.1. Unverschlüsseltes Protokoll

Zu Beginn wurde überlegt, ob auch ein unverschlüsseltes Protokoll ausgeführt werden soll. Dies war zum Beginn der Implementierungsphase möglich bis das Proof-of-Concept so angepasst wurde, dass es entweder `liboqs-python` oder `python-mbedtls`-Funktionen zum Verschlüsseln nutzt. Da das Ziel jedoch die Umsetzung eines quantensicheren Protokolls ist, wurde die Option auf ein unverschlüsseltes Protokoll entfernt.

6.1.2.2. Versenden der Nachrichten zwischen Server und Gerät

Das Erstellen der JSON-Tupel für den Versand der Nachrichten zwischen Gerät und Server funktioniert nur im Falle der 1:1-Beziehung dieser Implementierung (vgl. Abschnitt 4.4). Wenn mehr als ein Server und mehr als ein Gerät miteinander kommunizieren sollen, muss hier genau darauf geachtet werden, an welche Adressen die jeweiligen Nachrichten gesendet werden sollen. Es muss gegebenenfalls eine Prüfung der jeweiligen Absender- und Empfänger-Adressen durchgeführt werden.

Die generelle Erstellung und Verwendung mehrerer Instanzen von `Server` und `Device` sollte durch das Aktor-Framework prinzipiell möglich sein. Eine mögliche Umsetzung des Protokolls unter Verwendung mehrerer Geräte oder Server könnte in Bezug auf weiterführende Forschung durchgeführt werden.

6.1.2.3. Python

Python wurde als Programmiersprache gewählt, da sie sich für Prototypen-Implementierungen sowie Proof-of-Concepts eignet. Mit ihr lassen sich schnell Ergebnisse erzielen, die als Grundlage für weitere Tests und Forschung genutzt werden können. Z.B. wurde Python auch zur Programmierung des Servers (bzw. *Hubs*) beim Lazarus-Prototyp [35] [2] verwendet.

Ein Nachteil von Python ist die langsame Performance der darin geschriebenen Programme. Da es sich bei dieser Implementierung um eine Proof-of-Concept-Implementierung handelt, ist die absolute Laufzeit des Protokolls nicht relevant. Sie wurde trotzdem teilweise ausgewertet und es wurde festgestellt, dass die Performance mit maximal 4-5 Sekunden pro Szenario vermutlich schnell genug für die meisten Anwendungen ist. Dies müsste jedoch für jeden Anwendungsfall einzeln evaluiert werden.

In den Tabellen der Auswertung (Abschnitt 5.4.2 und 5.5.2) sowie im Anhang (Abschnitt A.5) werden die absoluten CPU-Zyklen aufgeführt. Die jeweiligen Vergleiche (vgl. Tabellen 5.8 und 5.16) zwischen maximalen und mininmalen Werten mit den Medianen und Mittelwerten werden mit Faktoren und Prozentangaben dargestellt, womit ein relativer Vergleich der CPU-Zyklen unter Verwendung der jeweiligen Algorithmen erreicht wird.

6.1.2.4. Aktorenmodell

Beim Watchdog-Timer-Protokoll in dieser Arbeit handelt es sich um ein verteiltes System, das aus einem Server und einem IoT-Gerät besteht. Das Aktorenmodell wurde gewählt, um die Nebenläufigkeit von mehreren Prozessen im Protokoll darstellen zu können. Dies wurde erfolgreich umgesetzt, da die Aktoren `Device`, `Timer` und `Sensor` zeitgleich Aufgaben ausführen können. Des Weiteren ist ein Hauptaspekt des Aktorenmodells, dass es auf dem Prinzip des Nachrichtenaustauschs zwischen verschiedenen Aktoren basiert. Auch der Nachrichtenaustausch wurde erfolgreich in der Implementierung umgesetzt, sodass verschiedene Nachrichten zwischen den Aktoren versendet werden.

Als Python-Aktor-Framework wurde Thespian gewählt, um eine reine Software-Implementierung des Proof-of-Concepts vorzunehmen. IoT-Geräte, welche mit einem Server kommunizieren, hätten - wie bei Lazarus [2] - auch auf Hardware-Ebene implementiert werden können. Eine weitere Möglichkeit wäre das Package und Aktor-Framework *Pykka* gewesen. Pykka ist das Python-Äquivalent zu Akka in Java. In weiterführender Forschung könnte ggf. ein Vergleich der Implementierung in beiden Frameworks durchgefürt werden.

In weiterführender Forschung sollte die Implementierung auf zwei Hostsystemen umgesetzt werden, um reale Anwendungen besser abzubilden. Auf jedem Hostsystem - sowohl Gerät als auch Server - muss eine Aktorsystem-Instanz gestartet werden. Getrennt wird die Ausführbarkeit der Aktoren, die nur auf einem der beiden Geräte ausgeführt werden sollen, durch sogenannte Capabilities, die an jeweilige Eigenschaften des Hostsystems gebunden sind. Die Capabilities der Aktoren werden dazu mit dem Decorator `@requireCapability()` [78] für jeden Aktor einzeln festgelegt. In dieser Arbeit wurden bei der Implementierung keine `@requireCapability`-Dekoratoren verwendet, da beide Hauptaktoren auf einem System laufen und somit für beide die gleichen Systemabhängigkeiten bestehen. Sollten Gerät und Server auf zwei unterschiedlichen Hostsystemen laufen, so sollten diese Dekoratoren verwenden werden, um zu markieren, dass Instanzen der Klassen `Device` und `Server` jeweils nur auf die Klassen zugreifen können, die mit dem jeweiligen Decorator ausgestattet sind. Das Gerät soll nicht auf Klassen des Servers zugreifen können und vice versa.

Ein weiterer Punkt des Nachrichtenversands im Aktorsystem, ist die Tatsache, dass jeder Aufruf der `send()`-Methode eines Aktors asynchron ist. Das bedeutet, dass es keine Garantie für eine immer gleiche Reihenfolge beim Empfangen von Nachrichten gibt. Werden nur zwei Aktoren betrachtet und A sendet mehrere Nachrichten an B, so erhält B die Nachrichten in der Reihenfolge wie sie von A gesendet wurden. Wenn allerdings C auch Nachrichten an B versendet, kann nicht garantiert werden, dass der Empfang der Nachrichten von A und C nicht miteinander vermischt werden. Außerdem können Aktoren immer nur eine Nachricht auf einmal verarbeiten. [79] Dies wurde bei den Szenarien 4 und 5 beobachtet, weshalb für diese Szenarien keine CPU-Zyklen gemessen wurden. Bei den anderen Szenarien wird aufgrund der Tatsache, dass Aktoren je nach Art der empfangenen Nachricht und je nach Sender unterschiedliche Aufgaben durchführen, indirekt eine Reihenfolge eingehalten. Teilweise kann es jedoch passieren, dass Aktoren eine `ActorExitRequest` erhalten, wodurch ein Aktor oder das Aktorsystem beendet werden und weitere Aktoren nicht mehr ausgeführt werden können und so ggf. die Reihenfolge nicht eingehalten wurde. Die Überprüfung der Reihenfolge wurde mithilfe von `print`-Statements manuell durch die Autorin dieser Arbeit durchgeführt. Die Überprüfung könnte bei Bedarf noch durch einen Logger, der jedes Senden- und Empfangen aufzeichnet, ergänzt werden. Die Implementierung sollte wie bereits beschrieben auf mindestens zwei Hostsystemen im Stil der weiteren Stufen der Technology Readiness Level getestet werden.

6.2. Diskussion der Messungen

In dieser Arbeit wurden zunächst die CPU-Zyklen bestimmt, die für die Funktionsaufrufe `gen_keypair()`, `sign()` und `verify()` zweier Python-Wrapper-Bibliotheken (Abschnitt 5.4.2) benötigt werden. Anschließend wurden in Abschnitt 3.1.4 die CPU-Zyklen der verschiedenen Szenarien (Abschnitt 5.5.2) unter Verwendung dieser Funktionen gemessen.

Die Messreihen der CPU-Zyklen wurden ausschließlich auf dem in Abschnitt 5.1.1 genannten Notebook durchgeführt. Die Performance der Funktionen und Szenarien wurde in CPU-Zyklen (Cycles) gemessen, da diese unabhängig vom verwendeten Prozessor sind. Um die CPU-Zyklen zu verifizieren, sollten die Messungen in Zukunft noch auf mindestens einem weiteren Rechner oder einem Server ausgeführt werden. Die CPU-Zyklen sollten dabei nicht grundlegend voneinander abweichen. Die Auswertung der Messreihen erfolgte durch zwei verschiedene nicht generisch aufgebaute Auswerteskripte (Abschnitt 5.3.4), die die mithilfe der Messskripte generierten `.csv`-Dateien der Messreihen auswerten. Bei Bedarf können die Auswertungen noch durch weitere statistische Methoden ergänzt oder die CPU-Zyklen mithilfe der Taktfrequenz des verwendeten Prozessors in Sekunden umgerechnet werden (siehe Anhang A.4).

Aus den Szenario-Messungen und den Funktionsmessungen geht hervor, dass Dilithium3 einer der performantesten Algorithmen der für die getesteten Szenarien ist.

6.2.1. Algorithmen und Bibliotheken

Bei den Messungen wurden 31 digitale Signaturalgorithmen berücksichtigt (vgl. Tabelle 5.3). Darunter sind drei der beim NIST eingereichten Algorithmen, die mit verschiedenen Parametersets ausgeführt werden können. In dieser Arbeit wurden 24 SPHINCS+-Parametersets getestet (vgl. Abschnitt 2.4.2.5), zwei Falcon- und zwei Dilithium-Parametersets, sowie drei klassische Verfahren.

Der Algorithmus *Dilithium2* wurde bei den Messungen nicht berücksichtigt, da es kein weiteres vergleichbares Parameterset auf demselben NIST-Level bei den gewählten Algorithmen gibt. Da bei der Auswertung der Messungen jedoch aufgezeigt wurde, dass auch manchmal Level 5-Parametersets schneller als Level 1-Parametersets sind, könnte Dilithium2 bei zukünftigen Messungen mitberücksichtigt werden. Die Parametersets von *SPHINCS+*, die mit der Hashfunktion *Haraka* arbeiten, wurden ausgeschlossen, da die Hashfunktion vom NIST bisher nicht genehmigt wurde. Da keine weiteren digitalen Signaturalgorithmen in die vierte Runde des Standardisierungsprozesses übernommen wurden [13], wurden in dieser Arbeit keine weiteren Signaturalgorithmen betrachtet.

Für die Durchführung der Messreihen wurden die in den Abschnitten 5.3.1 und 5.3.2 beschriebenen Messskripte und die Implementierung mit den Algorithmen der Bibliotheken `python-mbedtls` und `liboqs-python` (vgl. Tabelle 5.3) ausgeführt.

Es gibt noch weitere Bibliotheken, die jeweils einen der drei quantensicheren digitalen Signaturverfahren bereitstellen. Diese beruhen teilweise auf den `liboqs`-Implementierungen

und wurden auf GitHub bereitgestellt. `pydilithium` ist eine Bibliothek von Beechat Network Systems Ltd., die Dilithium2 implementiert und auf GitHub unter `dilithium-py` zu finden ist [80]. Die Bibliothek `falcon` von Prest, einem der Designer von Falcon, stellt die zwei relevanten Bitsicherheitsniveaus von 512 Bit und 1024 Bit bereit [81]. `pyspx` ist eine Bibliothek für SPHINCS+-Verfahren, die mit `pip` [82] installiert oder von GitHub geklont werden kann [83]. Die Packages wurden kurz getestet, es wurde sich jedoch gegen sie entschieden, da für jede Bibliothek unterschiedliche Datentypen als Parameter für die Funktionen benötigt wurden und somit vermutlich mehr Umrechnungen und Code nötig gewesen wären. Darunter hätte ggf. die Vergleichbarkeit gelitten.

Des Weiteren beinhaltet die Bibliothek `pqclean` [84] die drei quantensicheren Algorithmen. Falcon und Dilithium sind außerdem in der Bibliothek `pqm4` [85] vertreten. Um die Performance der in `pqclean` und `pqm4` implementierten Funktionen einzuordnen, könnten weitere Messungen und Auswertungen durchgeführt werden. Des Weiteren werden voraussichtlich noch weitere digitale Signaturverfahren beim NIST eingereicht, die getestet werden können. [13]

Prinzipiell ist darauf zu achten, dass die Funktionsaufrufe `gen_keypair`, `sign` und `verify` der Bibliotheken `python-mbedtls` und `liboqs-python` mit unterschiedlichen Datentypen als Parametern aufgerufen werden und auch unterschiedliche Datentypen als Parameter zurückgeben (siehe Datei `crypto.py`). Des Weiteren werden verschiedene Kontexte verwendet, die auf dem zugrundeliegenden Algorithmus basieren. Der Vergleich der klassischen und quantensicheren Funktionsaufrufe ist daher mit Vorsicht zu betrachten und ist auch relevant bei der Betrachtung der Performance bei den Szenarien.

Für die Schlüsselerstellung wird bei `python-mbedtls` ein Funktionsaufruf verwendet, bei `liboqs-python` sind es zwei Aufrufe. Dies erschwert außerdem den Vergleich der einzelnen `generate`-Funktionen. Eine Alternative wäre die zeitliche Bestimmung der gesamten Erstellung mit Exportierung der privaten und öffentlichen Schlüssel. `python-mbedtls` exportiert die Schlüssel direkt im `.der`-Format, wohingegen `liboqs-python` die Schlüssel in `Bytes` zurückgibt. Die Verarbeitung der Schlüssel in `sign` und `verify` erfolgt je nach verwendeter Bibliothek unterschiedlich. Die Speicherung der Schlüssel erfolgt auf dem Hostsystem prinzipiell im `.der`-Format. Signatur und Nachricht werden bei beiden Bibliotheken als `Bytes` ausgegeben und werden daher auch als Parameter des Typs `Bytes` bei `sign` und `verify` erwartet. Die Verifikationsfunktion gibt in beiden Fällen einen booleschen Wert zurück.

Die Signier- und Verifiziervorgänge der beiden Bibliotheken können nicht direkt miteinander verglichen werden, da die Signier- und Verifizierfunktionen mit unterschiedlichen Typen für die Schlüssel arbeiten.

Es wurden außerdem Messungen zu den benötigten CPU-Zyklen für die Schlüsselerstellung aufgenommen und in den Tabellen in Anhang A.5.1 dargestellt. Die Auswertung der Messwerte wird in dieser Arbeit nur betrachtet, um eine grobe Einschätzung der Dauer einer Schlüsselerstellung zu erhalten. Prinzipiell spielt dies für die Arbeit jedoch keine Rolle, da das Device Provisioning nicht als Teil des Protokolls angesehen wird und die Schlüsselerstellung und -verteilung nur einmal am Anfang durchgeführt wird. Im Fall des Lazarus-Prototyps erfolgt auch während des Protokolls eine neue Schlüsselerstellung auf dem Gerät.

Dies sollte nur sehr selten erfolgen, sodass die Performance nicht relevant ist. Bei einer Implementierung in der realen Welt sollte das Protokoll ggf. um die TEE erweitert werden, wodurch die Schlüsselerstellung und der Schlüsselaustausch im Gesamtprotokoll ergänzt werden. Dies ist relevant in Bezug auf die Erstellung eines neuen Geräte-Schlüsselpaars während der Laufzeit des Protokolls (vgl. DICE und DICE++ bei Lazarus [2]).

6.2.2. Ergebnisse der Funktionsmessungen

Es wurden für jede Messreihe 10.000 Messwerte aufgenommen, damit Ausreißer bei der Auswertung weniger ins Gewicht fallen und eine statistische Relevanz erzeugt wird.

Bei einem Blick auf die Benchmarks der C-Referenzimplementierungen der drei PQC-Algorithmen fällt auf, dass die neuen Messwerte (vgl. Abschnitt 5.4.2) von den Messwerten der C-Referenzen abweichen (vgl. [64] [81] [67]). Daher werden für die vorliegende Arbeit ausschließlich die in Abschnitt 5.4.2 aufgeführten Messwerte als Referenz angenommen. In der README.md [86] der verwendeten Referenzimplementierung von Dilithium wird beschrieben, dass bei der Referenzimplementierung auf sauberen Code und nicht auf Optimierung geachtet wurde und daher eigene Benchmarks von den Benchmarks der C-Referenzimplementierungen abweichen können. Ggf. wurden im Hintergrund Optimierungen vom Notebook durchgeführt, die nicht antizipiert wurden. Die Auswertung der Dilithium3 und Dilithium5-Funktionen liegt näher an den Messwerten, die mit optimiertem Code auf einem Skylake-Rechner erreicht wurden. Es wird vermutet, dass dies auch bei Falcon und SPHINCS+ der Fall ist. In weiterführender Forschung könnten die Python-Benchmarks mit den Benchmarks der C-Referenzimplementierungen verglichen werden und herausgearbeitet werden, warum diese voneinander abweichen.

Der schnellste quantensichere Algorithmus ist laut Ricci *et al.* [63] Dilithium3. Diese Aussage kann durch die Messreihen der Python-Wrapper-Funktionen zumindest im Vergleich zu allen Falcon- und SPHINCS+-Parametersets verifiziert werden (vgl. Abschnitt 5.4.2).

Die Falcon-Parametersets und Dilithium5 ist mit Dilithium3 am schnellsten bei den Funktionen `sign` und `verify`. Die Schlüsselerstellung benötigt bei Falcon-512 bzw. Falcon-1024 jedoch länger als bei Dilithium3. Da die Schlüsselerstellung nicht zum Gesamtprotokoll hinzugezählt wird, ist die dafür benötigte Zeit eher zweitrangig. Die Aussage von [66] lässt sich bestätigen: Falcon ist bei der Verifizierung schneller als bei der Signaturerstellung (vgl. Abschnitt 2.4.2.4). Im Falle der Funktionsmessungen ist Falcon-512 beim Verifizieren eines Signatuer ca. 3,8 Mal schneller als beim Erstellen. Falcon-1024 ist beim Verifiziervorgang ca. 4,4 Mal schneller (vgl. Tabellen 5.6 und 5.7)

6.2.2.1. Nachrichtenlängen

Bei den Messreihen wurden verschiedene Nachrichtenlängen (64 Byte, 96 Byte, 128 Byte) als Eingaben für die klassischen Signatur- und Verifizierfunktionen gewählt, allerdings wurden zeitgleich auch jeweils unterschiedliche Hashfunktionen (SHA256, SHA384, SHA512)

für diese Funktionen gewählt. Bei den Messreihen der quantensicheren Algorithmen wurden drei verschiedene Nachrichtenlänge für die gleichen Signatur- und Verifizierfunktionen gewählt, sodass hier verglichen werden kann, ob die Nachrichtenlänge eine Auswirkung auf die Anzahl der CPU-Zyklen hat. Ein Vergleich der Mittelwerte der jeweiligen Messreihen mit gleichem Algorithmus und unterschiedlichen Nachrichtenlängen (Tabelle 5.4 und Abschnitt 5.4.2.1) hat gezeigt, dass es bei den Nachrichtenlängen 64, 96 und 128 Byte kaum Abweichungen bei der Anzahl an CPU-Zyklen in Bezug auf die Länge der zu signierenden Nachrichten gibt. Durch die Messreihen kann nicht festgestellt werden, dass es vergleichsweise länger dauert eine Nachricht mit der Länge 128 Byte zu signieren als eine Nachricht der Länge 64 Byte. Dementsprechend wurde bei den Messreihen für die Szenarien angenommen, dass die Nachrichtenlänge bei der Signaturerstellung und -verifizierung nicht relevant ist. In zukünftiger Forschung sollten jedoch auch noch weitere Nachrichtenlängen bestimmt werden, die auch mehr Byte haben, um die Aussage auch für größere Nachrichten zu verifizieren.

6.2.2.2. Standardabweichung und maximale Werte

Es fällt auf, dass die Standardabweichungen der Messreihen bei allen drei Funktionen sehr hoch sind und die Mediane teilweise um bis zu 19,8 Prozent von den Mittelwerten abweichen. Die Standardweichung bei den Messreihen der Schlüsselerstellung zum Mittelwert liegt bei Faktor 27,9. Bei der Signaturerstellung weicht die Standardabweichung bis zu einem Faktor 75,8 vom Mittelwert ab, bei der Signaturverifizierung beträgt der Faktor der Abweichung nur 15,6.

In Abschnitt 5.4.2.5 wurde erläutert, dass die maximalen Werte bis zu 10 Mal höher sind als die Mediane und Mittelwerte. Im Vergleich zu den Abweichungen der maximalen Werte von den Medianen bzw. Mittelwerten bei den Szenario-Messungen (vgl. 5.5.2), bei denen die maximalen Werte nur bis zu 1,755 Mal höher liegen, ist dies ca. eine Verfünffachung.

Mögliche Gründe für die hohen maximalen Werte und Standardabweichungen könnten sein:

- Theorie 1: Randomisierung der Verarbeitungsreihenfolge innerhalb der Funktionen.
- Theorie 2: Die Schlüsselpaare lassen sich unterschiedlich schnell erstellen und es kann unterschiedlich lange dauern mit ihnen zu rechnen. Es gibt sozusagen „schwere“ Schlüssel.
- Theorie 3: Durch das verwendete Notebook spielte sogenanntes Throttling in die CPU-Zyklus-Messungen hinein.

Die *erste Theorie* begründet sich auf der Vermutung, dass die Designer nicht wollen, dass die Schlüssel leicht mit einem Seitenkanalangriff zu knacken sind. Gründe dafür können sein, dass die Verarbeitungsreihenfolge innerhalb der Funktionen zufällig gewählt wird, sodass weder ein Timing Oracle noch eine Energieanalyse die Schlüssel preisgeben könnte. Die Designer von Falcon beschreiben, dass eine bestimmte Berechnung in konstanter Zeit

erfolgen muss, damit die Algorithmen gegenüber Timing-Angriffen (eine Art von Seitenkanalangriff) resistent sind [71]. SPHINCS+-SHA2256 und -SHAKE256 verwenden keine auf Geheimnissen basierenden Lade- oder Speicheroperation und verarbeitet geheime Daten nur in symmetrischen kryptografischen Primitiven. Prinzipiell sind differentielle Angriffe und Fault-Angriffe auf SPHINCS+ möglich. Als Gegenmaßnahme besteht die Option die Signaturerstellung zu randomisieren. [67]

Theorie 2 begründet sich auf der Tatsache, dass die Messskripte zu den Funktionen jeweils aus nur einem Skript bestehen, in dem für jede Messung die Schlüsselpaare neu erstellt werden, mithilfe dessen eine Signatur erstellt und anschließend verifiziert wird. Für jeden Algorithmus wurde eine neue Nachricht generiert, die für alle 10.000 Messungen eines Algorithmus gleich blieb, aber die Schlüssel änderten sich, da die Schlüsselerstellung innerhalb einer Schleife stattfindet. Grund hierfür war die Bestimmung der Anzahl der CPU-Zyklen, die für eine durchschnittliche Schlüsselpaarerstellung benötigt werden. Dies ist hilfreich für die Übersicht, da der Fokus jedoch auf der Signaturerstellung und -verifizierung liegt, hat diese Übersicht keine große Relevanz.
Um Theorie 2 zu überpüfen, kann ein neues Messskript geschrieben werden, das aus zwei Teilen besteht. Zum Starten des Skripts wird der zu verwendende Algorithmus und die Anzahl der durchzuführenden Messungen vom Anwender festgelegt. Des Weiteren wird das Schlüsselpaar dort erstellt und in einem bestimmten Ordner gespeichert. Das Skript greift auf eine Klasse in einer weiteren Datei zu, indem nur die Schlüssel aus der jeweiligen Datei geladen werden und im Anschluss die CPU-Zyklen für Signaturerstellung und Verifizierung der Signatur einzeln gemessen werden.

Theorie 3 wird aufgestellt aufgrund der Tatsache, dass es diese hohen Abweichungen generell gibt und die 10.000-malige Durchführung des Beispielprogramms im Messskript direkt hintereinander durchgelaufen ist, ohne dass das Skript pausiert wurde. Die Durchführung der Messreihen hat je nach Algorithmus unterschiedlich lange gedauert. Die Durchführung des Messskripts der Funktionen unter der Verwendung des Algorithmus Falcon-512 mit 10.000 Schleifendurchläufen hat knapp 5 Minuten benötigt, wohingegen die SPHINCS+-Algorithmen und rsa4096 bis zu 4 Stunden für 10.000 Durchläufe benötigt haben.

Da bei den Messungen nur ein Notebook verwendet wurde, könnte das Problem am Throttling liegen. Beim Throttling wird die Frequenz der CPU gedrosselt, um die Temperatur des Notebooks bei Überhitzung zu senken. Dadurch wird die Abwärme, aber auch die Performance reduziert [87]. Da im Messskript keine Abkühlzeiten eingeplant wurden, ist es gut möglich, dass Throttling vom Notebook genutzt wird, um eine Überhitzung zu reduzieren. Bei der Durchführung der Messungen wurden jedoch CPU-Zyklen gemessen, die unabhängig vom Throttling gleich bleiben sollten.
Um Theorie 3 zu testen, können die Messreihen der Funktionen auf dem Server der THM ausgeführt und getestet werden. Alternativ könnte der Lösungsansatz von Theorie 2 auch auf dem THM-Server ausgeführt werden.

6.2.3. Ergebnisse der Szenario-Messungen

Für die Szenarien 4 und 5 (Abschnitt 3.1.4) wurden keine CPU-Zyklen gemessen, da dort mehrere Aktoren parallel zueinander laufen und sich gegenseitig Nachrichten senden, bei denen die Reihenfolge der Bearbeitung nicht nachvollziehbar ist. Die Ergebnisse wären jedoch nicht repräsentativ, da der Ablauf bei jedem Durchlauf unterschiedlich ist. Es wurden daher nur die anderen sechs Szenarien gemessen, da diese immer gleich ablaufen und somit reproduzierbar sind.

Bei den Szenario-Messungen wurden 100 Messungen pro Szenario und Algorithmus aufgenommen. 100 Messungen sind im Gegensatz zu den 10.000 Messungen der Funktionen weniger repräsentativ (vgl. Abschnitt 5.4.2). Grund für die reduzierte Anzahl an Messungen pro Messreihe ist die für die Ausführung benötigte Zeit. Die Messung von 100 mal sechs Szenarien benötigt ca. zwei Stunden.

Für die Szenario-Messungen gibt es keine vergleichbaren C-Referenz-Benchmarks, da die Szenarien aus dem neu konzipierten Watchdog-Timer-Protokoll herausgearbeitet wurden.

6.2.3.1. Standardabweichung und minimale sowie maximale Werte

Die maximalen Werte liegen bei den Szenarien nur bis zu 75,5 Prozent höher als die Mittelwerte und Mediane, und liegen somit näher an den Mittelwerten als die maximalen Werte bei den Funktionsmessungen. Die Standardabweichungen liegen maximal 10 Prozent von den Mittelwerten entfernt. Daher wird vermutet, dass bei den Szenarien durch die Festlegung der bei jeder Messung gleichen Startzustände, keine zu großen Abweichungen bei den maximalen Werten zustande kommen.

Eine weitere Vermutung wäre, dass die Hardware sich durch kleinere Abkühlphasen nicht zu stark erwärmt, da für jede Messung eines Szenarios das Aktorsystem hoch- und am Ende wieder heruntergefahren wird. Das Skript, in dem die CPU-Zyklus-Messungen stattfinden, wird dadurch beendet und wieder neu ausgeführt. Im Rahmen von weiteren Messungen könnten neben den bestehenden 100 Messungen, weitere Messungen ergänzt und zur Auswertung hinzugezogen werden, damit mögliche Ausreißer weniger in die Mittelwerte und Mediane hereinspielen. Allerdings weichen die Mittelwerte und Mediane bei den Szenario-Messungen bereits nur gering voneinander ab, wodurch die Messreihen als ausreichend angenommen werden.

Eine weitere mögliche Erklärung wäre die Tatsache, dass sich die minimalen und maximalen Werte für einzelne Berechnungen durch die Vielzahl an durchgeführten Berechnungen während der Szenarien mitteln und Ausreißer dadurch nicht auffallen.

6.2.3.2. Schnellste Algorithmen

Aus der Auswertung (Abschnitt 5.5.2) geht hervor, dass keiner der Algorithmen für jedes Szenario im Vergleich zu den anderen Algorithmen weniger CPU-Zyklen benötigt. Die Tendenz zur Wahl eines Algorithmus in Bezug auf die Performance liegt bei Dilithium3, einem der beiden Falcon-Parametersets oder Dilithium5. In der Regel benötigen die Algorithmen Falcon und Dilithium ähnlich viele CPU-Zyklen und sind schneller als SPHINCS+, mit Ausnahme von den Szenarien 3a und 7. Bei Szenario 3a und 7 muss keine Signatur erstellt werden und es wird keine Nachricht über das Netzwerk versendet, wodurch die Abweichungen der Mittelwerte mit maximal 16,4 bzw 17,8 Prozent sehr gering sind. Obwohl Dilithium3 der Algorithmus ist, der am schnellsten Verifizieren kann, ist bei diesen beiden Szenarien SPHINCS+-SHAKE128s-simple am schnellsten. Aus diesem Grund sollte auf Basis der jeweiligen Performance der Szenarien abwogen werden, welcher Algorithmus für den Anwendungsfall am geeignesten ist.

Des Weiteren fällt auf, dass das NIST-Sicherheitslevel keinen gleichmäßigen Einfluss auf die Performance der Algorithmen hat. Bei den Szenarien 1, 3b und 7 sind die Level 1 Parametersets schneller als die für Level 3 und 5. Dilithium3 benötigt bei Szenario 2 weniger CPU-Zyklen als die Parametersets für Level 1 und 5.

Aus den Tabellen der Szenarien (Abschnitt 5.5.2) geht hervor, dass sich die Performance klassischer und quantensicherer Algorithmen nicht stark voneinander unterscheidet, da die Falcon- und Dilithium-Parametersets ähnlich viele CPU-Zyklen für die Ausführung der Szenarien benötigen. Da die quantensicheren Verfahren ähnlich lange dauern wie die klassischen Verfahren, ist in Bezug auf die Performanz die Wahl eines quantensicheren Algorithmus kein Nachteil. Das Protokoll dauert ähnlich lange, hat aber ein höheres Sicherheitslevel.

Im Nachfolgenden werden Gründe für die unterschiedliche Performance der Algorithmen in Bezug auf die Szenarien genannt, die in weiterführenden Betrachtungen erforscht werden können.

Nachrichtenlängen. Die Nachrichtenlängen (64, 96 und 128 Byte), die bei den Funktionsmessungen getestet wurden, sind kleiner als die Payloads (Nachrichten), die tatsächlich im Protokoll signiert oder verifiziert werden (zwischen 106 und 210 Byte). Dadurch kann das Signieren und Verifizieren (inklusive Hashen) sowie das Packen und Entpacken von JSON-Strings zeitlich variieren. Für weitere Messungen der Funktionen sollten die tatsächlich im Protokoll genutzten Nachrichtengrößen verwendet werden. Da die im Signaturalgorithmus integrierte Hashfunktion bei der Berechnung der Hashes der im Protokoll genutzten Nachrichten mehr Zeit benötigen könnte als bei den in Abschnitt 5.4.1 verwendeten Nachrichtenlängen.

Des Weiteren muss bei Updates in einer realen Anwendung die Datei zunächst gehasht werden, da sie sonst nicht direkt signiert werden kann. Bei einigen Signaturalgorithmen - wie auch bei den in dieser Arbeit genutzten - sind jedoch die Hashfunktionen integriert.

Um Daten über das Netzwerk versenden zu können, werden diese beim Watchdog-Timer-Protokoll ins JSON-Format umgewandelt. In weiteren Messreihen könnten auch die Größen, der über das Netzwerk zu versendenden Daten überprüft werden, um einen Netzwerk-Overhead feststellen zu können. Dazu müsste das Protokoll jedoch mit zwei Hostsystemen arbeiten.

Verwendete Startzustände. Für jeden Algorithmus wurden eigene Schlüssel, mithilfe des gewählten Signaturalgorithmus, erstellt und mit diesen wurden Noncen, Boottickets, Updates und Versionssnummern zum Herstellen der gleichen Startzustände für alle 100 Messungen eines Szenarios generiert. Die Schwierigkeit der Startzustände könnte sich sowohl positiv als auch negativ auf die Performance auswirken.

Dauer der Szenarien. Die Szenarien führen unterschiedlich oft die Funktionen `sign` und `verify` aus und versenden Daten über das Internet an den jeweils anderen Kommunikationspartner, außerdem werden vor dem Versenden von `Request`s Noncen mit einem HRNG generiert, was ggf. auch unterschiedlich viele Berechnungen in Anspruch nehmen kann.

In Tabelle 6.1 ist die Häufigkeit, der in den Szenarien verwendeten Funktionen, aufgeführt. Die Dauer der jeweils betrachteten Szenarien ist gering, wenn die CPU-Zyklen in Sekunden umgerechnet werden. Somit ist die Verwendung des Protokolls für die Anwendung in der realen Welt nutzbar.

Die unterschiedliche Performance der Algorithmen in Bezug auf die Dauer der einzelnen Szenarien könnte an den benötigen CPU-Zyklen der Funktionsaufrufe liegen, die aufsummiert werden. Manche Algorithmen sind im Gesamtbild schneller im Verifizieren als im Falsifizieren von Signaturen. Das Falsifizieren wurde bei den Funktionsmessungen nicht betrachtet, könnte aber mit weiteren Messreihen evaluiert werden.

Am Beispiel von Falcon-512, Dilithium3 und SHAKE256-192s-robust lässt sich erahnen, dass die Anzahl der Funktionen bei den einzelnen Szenarien relevant ist (vgl. Tabelle 6.1). Bei den drei Algorithmen benötigen die Szenarien 2 und 6 am längsten, da dort die meisten relevanten Funktionen aufgerufen werden. Des Weiteren benötigen die Szenarien 3a und 7 bei allen Algorithmen in etwa gleich lange. Die Vermutung liegt nahe, dass die Falsifizierung von Signaturen schneller ist als die erfolgreiche Verifizierung bzw. die zusätzliche Signierung von Nachrichten. SPHINCS+-SHAKE256-192s-robust ist sehr viel schneller beim Verifizieren von Signaturen (Szenarien 3a und 7) als bei den anderen Szenarien. Grund dafür könnte sein, dass bei diesem Algorithmus das Signieren ca. 507 Mal länger dauert als das Verifizieren. Dies spiegelt sich in den benötigen CPU-Zyklen wider, da bei den Szenarien 1, 2 und 6 jeweils zweimal signiert wird und bei den Szenarien 3 b und 8 jeweils einmal. Durch einen Vergleich der CPU-Zyklen für das Signieren und Verifizieren unter Verwendung des Algorithmus wird deutlich, dass das Signieren einen sehr großen Teil der CPU-Zyklen bei den Szenarien benötigt. Grund dafür könnte sein, dass das alleinige Signieren ca. genauso viele CPU-Zyklen im Mittel benötigt als für die jeweilige Ausführung des gesamten Szenarios. Hierbei sollte jedoch nochmal darauf aufmerksam gemacht werden, dass die Nachrichtenlängen der signierenden Nachrichten bei den Messungen der Funktionen geringer waren

und somit eine eindeutige Abbildung der CPU-Zyklen der einzelnen Funktionen auf die der Szenarien nicht möglich ist.

Szenario	Anzahl `sign`	Anzahl `verify`	Anzahl HRNG	Anzahl Versand	Anzahl insgesamt	Falcon-512 (in Mrd. Zyklen)	Dilithium3 (in Mrd. Zyklen)	SPHINCS+-SHAKE256-192s-robust (in Mrd. Zyklen)
1	2	1	1	2	6	0,35	0,37	10,25
2	2	2	1	2	7	0,42	0,41	10,66
3a	-	1	-	-	1	0,20	0,20	0,21
3b	1	1	0	1	3	0,30	0,31	5,56
6	2	2	1	2	7	0,42	0,42	10,62
7	-	1	-	-	1	0,25	0,25	0,25
8	1	-	-	1	2	0,22	0,22	5,18

Tabelle 6.1.: Anzahl bestimmter Funktionen bei den einzelnen Szenarien.

Durch die Betrachtung der durchgeführten Funktionen bei den Szenarien benötigen die Algorithmen relativ gesehen ähnlich viele CPU-Zyklen. Aus der Häufigkeit der Anzahl der genannten Funktionsaufrufe und Versände insgesamt lässt sich die ungefähre relative Dauer der einzelnen Szenarien ablesen.

Im Folgenden wurden die absoluten CPU-Zyklen mithilfe der CPU-Frequenz des für die Messungen verwendeten Notebooks in Sekunden umgerechnet. Beim langsamsten Algorithmus werden zum Anfragen eines neuen Updates (Szenario 6) sowie zum Anfragen eines neuen Boottickets (Szenario 2) maximal 4 Sekunden benötigt. Die Verifizierung eines vorliegenden Boottickets (Szenario 3a) oder Updates (Szenario 7) dauert im Vergleich dazu nur maximal 0,11 Sekunden. Die Vermutung liegt nahe, dass die hohen CPU-Zyklen durch die Erstellung einer Anfrage mit dem HRNG bzw. die Anzahl der benötigten Signierfunktionen (vgl. Tabelle 6.1) oder die Validierung der Nonce auf `Device`-Seite zu stande kommen, da die Verifizierung auf `Server`-Seite - wie an Szenario 3 und 7 zu erkennen ist - nicht sehr lange dauert.

Bei Messungen der Szenarien 4 und 5 hätte herausgefunden werden können, wie viele CPU-Zyklen das `Device` benötigt, um eine Deferralticket-Anfrage zu stellen und zu beantworten. Diese Frage könnte in weiteren Messungen durch die Aufteilung der Szenarien 4 und 5 auf kleinere Abschnitte ermöglicht werden. Die Erstellung von Update- und Bootticket-Anfragen dauert ähnlich lange, obwohl bei Updates keine Nonce erstellt wird. Szenario 8 benötigt wenig Zeit zum Erstellen und Senden eines Updates.

6.2.3.3. Doppeltes Hashen vor den Funktionen sign und verify

Bei einer digitalen Signatur wird vor dem Verschlüsseln mit dem privaten Schlüssel ein Hash, der anschließend verschlüsselt wird, generiert. Bei den in der Implementierung verwendeten Bibliotheken ist eine Hashfunktion in den `sign`- bzw. `verify`-Funktionen integriert. Deshalb ist keine explizite Angabe eines Hashalgorithmus notwendig.

In einer vorherigen Version der Implementierung wurden die zu versendenden und signierenden Nachrichten jedoch vor der `sign`- bzw. `verify`-Funktion mit der Hashfunktion SHA512 gehasht und der daraus entstandene Hash wurde als Nachricht von der `sign`- bzw. `verify`-Funktion entgegengenommen. Des Weiteren wurde der Hash mit der Nachricht mitgesendet, sodass er vom Empfänger mit dem eigens generierten Hash der Nachricht verglichen werden konnte. Dieser Vergleich wurde in der Implementierung mithilfe einer Vergleichsfunktion umgesetzt. Das Hashen von Dokumenten vor dem Signieren ist eine Common Practice, da sie sonst zu groß für den Signaturalgorithmus sind. Es ging zunächst nicht eindeutig aus der Bibliothek hervor, dass die Hashfunktion im Signaturalgorithmus integriert ist, weshalb bereits Szenario-Messungen mit zusätzlichem Hashen und einem Vergleich von Hashes durchgeführt wurden. Ein weiterer Grund für die Implementierung einer Hashfunktion vor dem Signieren war, damit die Nachrichten, die den Funktionen als Parameter mitgegeben werden gleich lang sind, womit die Signierfunktion kaum variierende CPU-Zyklen benötigen sollte. Dies kann aus den mit dieser Implementierung durchgeführten Messreihen nicht herausgelesen werden, da zusätzlich zum Signieren noch einige weitere Funktionen bei den jeweiligen Szenarien mit in die Anzahl der Zyklen hineingezählt werden.

Für beide Implementierungen wurden die CPU-Zyklen der Szenarien gemessen, wobei nur die Messauswertung ohne vorheriges Hashen in dieser Arbeit beschrieben und interpretiert wird. Es wurde vermutet, dass die Szenarien mit zusätzlichem vorherigen Hashen der Nachricht jeweils einige Zyklen mehr Zeit benötigen als die ohne vorheriges Hashen. Durch einen Vergleich der beiden Auswertung je Szenario wurde allerdings festgestellt, dass mit vorherigem und sozusagen doppeltem Hashen die Performance einiger Algorithmen besser ist. Ein Beispiel wäre, dass der Algorithmus Falcon-1024 bei allen Szenarien schneller ist als alle anderen Algorithmen. Falcon-1024 benötigt ohne vorheriges Hashing bis zu ca. 10 Prozent länger als mit Hashing. Bei den Messreihen mit zusätzlichem Hashing vor den Funktionen `sign` und `verify` sowie dem zusätzlichen Prüfen des Hashes vor `verify` fällt auf, dass die CPU-Zyklen mal höher und mal niedrgier sind, als die ohne Hashing vor den Funktionen. Falcon-512 hingegen ist ohne vorheriges Hashing schneller. Dilithium3 und Dilithium5 sind ohne Hashing bis zu ca. 10 Prozent schneller als mit vorherigem Hashing.

Es werden nur Vermutungen für die Gründe aufgeführt, die jedoch im Rahmen dieser Arbeit nicht näher betrachtet werden und für weiterführende Forschungen genutzt werden können:

- Die Algorithmen arbeiten intern unterschiedlich mit abweichenden Parametern wie der Nachrichtenlänge, Blockgröße bzw. Hashlänge, weswegen mal die Algorithmen mit vorherigem Hashen und mal ohne vorheriges Hashen schneller sind.

- Da pro Messreihe neue Schlüsselpaare erstellt wurden, könnte es sein, dass mit einigen Schlüssel „schwerer“ zu rechnen bzw. zu signieren oder zu verifizieren ist als mit anderen.
- Das Hashen in der `sign`-Funktion benötigt länger für die ungehashten Nachrichten, da sie meist größer sind als 64, 96 oder 128 Byte.
- Das Hashen in der `sign`-Funktion benötigt länger für die ungehashten Nachrichten, weil deren Bytegröße keine Zweierpotenz ist.
- Das Hashen vor der `sign`-Funktion nimmt manchen Hashfunktionen in den `sign`-Funktionen einen größeren Aufwand ab.

Bei den Szenario-Messungen der klassischen Algorithmen wurde vor der `sign`-Funktion die gleiche Hashfunktion ausgeführt wie in der `sign`-Funktion: sha256 + sha256 oder sha384 + sha384 oder sha512 + sha512. Bei den quantensicheren Verfahren werden zwei unterschiedliche Hashfunktionen ausgeführt (sha512 + shake256 oder sha512 + sha256 oder sha512 + shake128). Daher sollte evaluiert werden, ob durch mehrmaliges Hashen mit gleichen (bei der Verwendung von `python-mbedtls`) oder zwei verschiedenen Hashfunktionen (bei der Verwendung von `liboqs-python`) die Sicherheit der Nachricht bzw. des Algorithmus gefährdet ist.

Laut der Spezifikation von SPHINCS+ [67] hat SPHINCS+-SHAKE unter Verwendung von SHAKE256 eine interne „Kapazität“ von 512 Bits. Bei der Verwendung von SHA2-256 in SPHINCS+-SHA256 werden nur 256 Bits als „chaining value“ verwendet. Dadurch kann der Algorithmus unter Verwendung von SHA2-256 in manchen Bereichen schwächer sein als SHAKE256 mit einer 256 Bit-Ausgabe. Daher wird manchmal SHA2-512 verwendet, um das gewünschte Sicherheitslevel zu erreichen. Bei SHA256 wird eine Nachricht von 512 Bit erwartet. Bei SHA384 und SHA512 wird eine Nachricht von 1024 Bit erwartet, wohingegen z.B. SHA3-256 eine Eingabelänge von 1088 Bit erwartet.

Falcon-1024 ist mit zusätzlichem vorherigen Hashen grundsätzlich schneller als ohne das vorherige Hashen. Dies trifft nicht auf alle Algorithmen zu, aber noch auf einen Großteil der SPHINCS+-SHA256 des Level 5. Ein ausführlicher Vergleich der Messergebnisse mit und ohne vorheriges Hashen liegt außerhalb des Rahmens dieser Arbeit. Daher sind Szenario-Messungen mit vorherigem Hashen nicht im Anhang der Arbeit, sind jedoch in der Abgabedatei enthalten. In weiterführender Forschung könnte überprüft werden, ob dies die Performance von bestimmten Algorithmen in bestimmten Szenarien verbessern kann.

6.3. Möglichkeiten der Anwendung im Internet of Things

Das konzipierte und implementierte Watchdog-Timer-Protokoll soll im Bereich IoT Anwendung finden. Dazu muss zunächst evaluiert werden, ob die auf dem Markt vorhandenen IoT-Geräte, die Eigenschaften besitzen, die durch die Verwendung quantensicherer digitaler Signaturverfahren gefordert werden. Ein IoT-Gerät hat - wie in der Einleitung (Kapitel 1)

beschrieben - verschiedene Einschränkungen, wenn es um Performance und Speicherplatz geht.

Die Tendenz liegt bei der Performance bei Dilithium3 oder einem der beiden Falcon-Sets sowie Dilithium5 (vgl. Abschnitte 6.2.2 und 6.2.3).

6.3.1. Speicherplatzbedarf

Der Speicherplatzbedarf der Implementierung gliedert sich in verschiedene Bereiche. Sowohl der Flashspeicher als auch der RAM sind bei IoT-Geräten ressourcenbeschränkt, sollen jedoch Updates empfangen und installieren, sowie Signaturen und Schlüssel erstellen und speichern können. Bei dem in dieser Arbeit betrachteten 1:1-Beziehungsbeispiel von einem Gerät zu einem Server wird jeweils das eigene Schlüsselpaar und der öffentlichen Schlüssel des Servers im Flashspeicher des Geräts gespeichert. Die Signaturen werden im RAM erstellt bzw. verifiziert und sollten daher auch dort hineinpassen bzw. streamingfähig sein - wie z.B. in Paper [23] am Beispiel von SPHINCS-256 beschrieben. Im Flashspeicher werden neben den Schlüsseln und ggf. Signaturen auch die Business-Logik und die Bootskripte gespeichert. Im Verlauf des Protokolls kann allerdings davon ausgegangen werden, dass parallel mehrere Signaturen erstellt oder verifiziert werden können, sodass für ausreichend RAM gesorgt werden sollte.

Bei einer Anwendung in der realen Welt müssten die Größen der versendeten Nachrichten neu evaluiert werden. Vorallem die Größe von Updates könnte bis in den Kilobyte-Bereich steigen, im Vergleich zu den in der Implementierung genutzten ca. 200 Byte. In der realen Welt müsste außerdem überprüft werden, wie groß solche Updates sein dürfen, damit sie in den Flashspeicher des IoT-Geräts passen.

In Tabelle 6.2 wird aufgeführt, wie viel Speicherplatz die Schlüssel der einzelnen Algorithmen benötigen, wenn davon ausgegangen wird, dass zwei öffentliche Schlüssel und ein privater Schlüssel gespeichert werden sollen. Der Speicherbedarf für Signaturen ist bei Dilithium und SPHINCS+ größer als bei den öffentlichen Schlüsseln. Da nicht bekannt ist, ob die Signaturen auf dem IoT-Gerät gespeichert werden müssen, konnte kein Speicherbedarf für die Signaturen festgelegt werden. Dies müsste je nach Anwendungsfall evaluiert werden. Die Erstellung der Signaturen findet jedoch teilweise auf dem Gerät statt, sodass dafür genügend RAM zur Verfügung stehen muss. Hülsing *et al.* [23] haben in ihrer Veröffentlichung geschrieben, dass es möglich ist eine 41 KB Signatur mit SPHINCS in 16 KB RAM zu erstellen. Dafür wurde ein ARM Cortex M3 mit einer Clock Speed von 32 MHz verwendet. Außerdem wurde für SPHINCS+ [22] bereits getestet, ob die Signaturen auf eingebetteten Systemen erstellt werden können [23], wobei dies für jedes IoT-Gerät, welches für das Watchdog-Timer-Protokoll in Betracht gezogen wird, überprüft werden sollte.

Die Tendenz zur Wahl eines Algorithmus liegt jedoch bei Dilithium3 oder einem der beiden Falcon-Sets, bei denen zwar die Schlüssel im Vergleich zu denen von SPHINCS+ größer sind, jedoch sind die Signaturen bei SPHINCS+ als bei Dilithium und Falcon.

Algorithmus	Public Key Device (Bytes)	Public Key Server (Bytes)	Private Key Device (Bytes)	Benötigter Speicher (Bytes)
Falcon-512	897	897	2000	3.794
Falcon-1024	1.793	1.793	2.000	5.586
Dilithium3	1.952	1.952	4.000	7.904
Dilithium5	2.592	2.592	4.864	10.048
SPHINCS+ 128	32	32	64	128
SPHINCS+ 192	48	48	96	192
SPHINCS+ 256	64	64	128	256

Tabelle 6.2.: Benötigte Ressourcen für Schlüssel auf dem Gerät, angegeben in Bytes. Die privaten Schlüsselgrößen von Dilithium wurden aus den Signer-Details der liboqs-Bibliothek, die bei den Durchführungen der Messungen gespeichert wurden, entnommen.

Bevor das Protokoll in IoT-Geräten implementiert werden kann, muss evaluiert werden, ob der Speicherplatz für die Implementierung selbst, die zu erstellenden Schlüssel und Signaturen sowie die zu installierenden Updates ausreicht. Die Umsetzung der weiteren Stufen der Technology Readiness Level müsste dafür in weiterführender Forschung erfolgen.

6.3.2. Sicherheit

Die *simple*-Parametersets von SPHINCS+ sind ca. doppelt so schnell wie die *robust*-Parametersets, haben aber einen Sicherheitsnachteil in Bezug auf ein *Random Oracle Model* und es sollte je nach Anwendungsfall das Sicherhitsrisiko abgewogen werden [67]. Auch aus den Auswertungen zu den einzelnen Funktionen und den Szenarien in den Abschnitten 5.4.2 und 5.5.2 geht hervor, dass die simple-Varianten meist weniger Zeit benötigen als die robust-Varianten.

Die Sicherheit bei der Verwendung eines Algorithmus, der dem NIST-Sicherheitslevel 5 zugeordnet ist, kann Dilithium oder Falcon verwendet werden. In den meisten Fällen sind diese schneller im Vergleich zu den SPHINCS+-Parametersets. Die Level 1 Algorithmen haben keine größeren Vorteile gegenüber den anderen zwei Leveln, außer, dass die Signaturen und Schlüssel kleiner sind.

Eine weitere Alternative zu einem ausschließlich quantensicheren Algorithmus ist die hybride Implementierung von einem klassischen und einem quantensicheren Algorithmus, die beide gebrochen werden müssten, um die Schlüssel zu berechnen. Dieser Ansatz könnte in zukünftiger Forschung umgesetzt und evaluiert werden.

6.3.3. Mögliche Anwendungsfälle

Das Watchdog-Timer-Protokoll kann für die Projekte SecDER [14] und IMMUNE [15] in Erwägung gezogen werden (vgl. Abschnitt 1.2). Dafür müsste evaluiert werden, wie viel Speicherplatz die jeweils verwendeten IoT-Geräte haben und wie häufig Daten vom Gerät

an den Server gesendet werden. Des Weiteren müssen die Zeiten für die `WakeupMessage`s sowie die Zeit des Watchdog-Timers bis zum Reset des Geräts festgelegt werden. Auch das gewünschte einzuhaltende Sicherheitslevel sollte dabei betrachtet werden.

Prinzipiell sollte für jeden möglichen Anwendungsfall genau evaluiert werden, welche Eigenschaften das Protokoll erfüllen sollte. Dementsprechend wird der Algorithmus entweder auf Basis der gewünschten Eigenschaften des Protokolls oder des IoT-Geräts gewählt.

6.4. Diskussion der verwendeten Methodik

Die verwendeten Forschungsmethoden waren eine geeignete Vorgehensweise zum Beantworten der Forschungsfragen (vgl. Abschnitt 7.3). Die induktive Vorgehensweise bei der Implementierung sowie der Durchführung der Messungen hätte auch deduktiv erfolgen können. So hätten zu Beginn der Bearbeitung Hypothesen bzw. Theorien aufgestellt werden müssen.

Jedoch konnten die gestellten Forschungsfragen mit der induktiven Vorgehensweise beantwortet werden und aus den Antworten konnten Theorien bzw. Vermutungen abgeleitet werden. Die Vermutungen wurden für jeden betrachteten Aspekt in den Abschnitten 6.2 und 6.3 aufgeführt und begründet.

Die Implementierung des Watchdog-Timer-Protokolls erfolgt qualitativ und die Durchführung der Messungen besteht aus mehreren Fällen bzw. Szenarien und ist somit quantitativ und kann Mehrfachfallstudie genannt werden. In dieser Arbeit wird die Mehrfachfallstudie jedoch als Experiment bezeichnet. Die Auswertungen der Messreihen der Funktionen sowie Szenarien sind quantitativ, da sie mithilfe statistischer Methoden erfolgt. Nachfolgend wird reflektiert, ob und wie die Gütekriterien bei der Implementierung des Watchdog-Timer-Protokolls (Abschnitt 6.4.1) und bei der Durchführung der Messungen (Abschnitt 6.4.2) eingehalten wurden.

6.4.1. Gütekriterien der Implementierung

In [41] werden Gütekriterien für qualitative Forschung aufgeführt, die zwar nicht standardisiert sind, aber von Genau [41] als sinnvoll erachtet werden. Diese Kriterien basieren auf Literatur der qualitativen Sozialforschung, werden im Folgenden jedoch auf die Implementierung des Watchdog-Timer-Protokolls übertragen.

- *Transparenz*: Es wird beschrieben, woher die Informationen zu den Algorithmen sowie die relevanten Bibliotheken (inklusive Versionsnummern) der verwendeten Algorithmen kommen. Es wird beschrieben, wie das System konzipiert, aufgebaut und implementiert wurde. Die für das Protokoll geschriebenen Klassen und Programme sowie Mess- und Auswerteskripte sind auf der beigelegten CD zu finden (siehe Auflistung der Inhalte in Anhang A.3). Die Erstellung der Tabellen der Auswertungen und ihre Inhalte werden begründet dargestellt.

- *Intersubjektivität*: Die Abläufe des Protokolls werden mithilfe von `print`-Statements im Terminal dargestellt, sodass jeder Benutzer überprüfen kann, ob das Szenario so durchgelaufen ist, wie es in der Konzeption in den Abschnitten 3.1.2 und 3.1.4 erläutert wurde.
- *Reichweite*: Die erneute Ausführung der Applikation mit gleichen Startzuständen (die Dateien und Schlüssel für die jeweiligen Algorithmen liegen im `temp`-Ordner vor) sollte zu gleichen bzw. ähnlichen CPU-Zyklen bei den Szenarien des Protokolls führen.

6.4.2. Gütekriterien der Durchführung der Messreihen

Die Einhaltung der folgenden Gütekriterien wurde mithilfe der Gütekriterien für ein gültiges Experiment [42], das auch auf die Durchführung von Messreihen ausgelegt werden kann, überprüft.

- *Validität*: Die Ergebnisse der Messreihen sind gültig, da sie die benötigten CPU-Zyklen für entweder die Funktionen 5.4.2 oder die Szenarien 5.5.2 messen.
- *Reliabilität*: Die Messungen sind reproduzierbar, da die verwendete Hardware beschrieben und die verwendete Software der Arbeit als CD beigelegt wurde. Die Installationsanleitung der Software (siehe Anhang A.1), sowie die verwendeten Mess- und Auswerteskripte liegen der Arbeit bei. In Kapitel 5 wird beschrieben wie die Messreihen durchgeführt wurden. Für die PQC-Algorithmen wurden die Commits in den signer.details gespeichert, sodass die exakte Version des verwendeten Algorithmus nachvollzogen werden kann. Des Weiteren sind `requirements.txt` der CD bzw. der Implementierung beigefügt.
- *Variierbarkeit*: Die gleichen Messreihen können mit unterschiedlichen Variablen wiederholt werden. Die möglichen Variablen sind in Tabelle 5.2 als Eingabeparameter für die Skripte aufgeführt.
- *Objektivität*: Die Ergebnisse der Messreihen können unabhängig vom Versuchsleiter genauso ermittelt werden. Es ist dabei jedoch zu beachten, dass die Messwerte aufgrund von im Hintergrund ausgeführten Programmen oder der verwendeten Hardware abweichen können. Die Messwerte sollten auf einem anderen Rechner jedoch nicht stark von den bereits erhaltenen Messwerten abweichen.
- *Planbarkeit*: Die Messungen können zu einem beliebigen Zeitpunkt durchgeführt werden.

Ein Nachteil der Methodik des Experiments bzw. der Durchführung von Messreihen ist der Zeitaufwand. Die Umgebung wurde künstlich hergestellt und ist nicht repräsentativ für die Anwendung in der realen Welt. In der realen Welt sollten Server und Gerät auf zwei separaten Geräten ausgeführt werden und über ein Netzwerk kommunizieren. Dies könnte in weiterführender Forschung z.B. im Zusammenhang mit den weiteren Schritten der Technology Readiness Level überprüft werden.

7. Fazit

In diesem Kapitel werden als Erstes die geleisteten Beiträge (Abschnitt 7.1) aufgeführt. In Abschnitt 7.2 werden die Ergebnisse der Arbeit und der Messungen zusammengefasst. Anschließend werden in Abschnitt 7.3 die Forschungsfragen beantwortet und es wird auf die erreichten Forschungsziele eingegangen. In Abschnitt 7.4 wird ein Ausblick auf mögliche zukünftige Forschung gegeben.

7.1. Geleistete Beiträge

Mithilfe der von der Autorin geleisteten Beiträge konnte ein Großteil der in Abschnitt 1.4.1 aufgeführten Forschungsziele erreicht werden. Um das Thema in den aktuellen Forschungsstand einordnen und die Grundlagen der Arbeit darstellen zu können, wurde daher eine Literaturrecherche durchgeführt. Anschließend wurden Algorithmen zum Testen auf Basis des NIST-Standardisierungsprojekts ausgewählt und nach Python-Bibliotheken recherchiert, die diese digitalen Signaturalgorithmen bereitstellen.

Im Zuge dieser Arbeit wurden Bausteine bestehender Protokolle evaluiert und mithilfe dieser ein generisches Watchdog-Timer-Protokoll für IoT-Geräte auf der Grundlage bestehender Forschung [2][3] konzeptioniert. Aufbauend auf dem entwickelten Konzept wurde eine Proof-of-Concept-Implementierung in Python umgesetzt, die sowohl mit klassischen als auch quantensicheren Signaturalgorithmen ausgeführt werden kann. Die Implementierung basiert auf einem Aktorenmodell und setzt die Funktionalitäten *Signieren* und *Verifizieren* um. Es wurden Messskripte geschrieben, um die Durchführung von CPU-Zyklus-Messungen von Funktionsaufrufen der Signaturalgorithmen zu ermöglichen. Des Weiteren wurde ein Messskript mit verschiedenen möglichen Anfangszuständen zum Messen von CPU-Zyklen festgelegter Szenarien unter Verwendung unterschiedlicher aber festgelegter Signaturalgorithmen geschrieben. Die jeweiligen Messungen der CPU-Zyklen der Funktionen und Szenarien wurden durchgeführt und mithilfe zweier selbstgeschriebener Auswerteskripte ausgewertet. Außerdem wurde eine Diskussion zu den erhaltenen Ergebnissen, sowie die Umsetzung der Methodik, des Protokolls und der Implementierung durchführt.

7.2. Zusammenfassung der Ergebnisse

In dieser Arbeit wurde eine Proof-of-Concept-Implementierung eines Watchdog-Timer-Protokolls umgesetzt, die in drei verschiedenen Versionen vorliegt und genutzt werden kann. Die Anleitungen sind in Anhang A.1 und A.2. Die Implementierung kann sowohl

mit ausgewählten klassischen sowie quantensicheren Signaturalgorithmen ausgeführt werden. Die Implementierung kann, je nachdem welcher Startzustand vorliegt, unterschiedlich ablaufen. Die in der Konzeption (Kapitel 3) herausgearbeiteten Szenarien wurden, unter Einhaltung der Schutzziele der jeweils gesendeten Nachrichten, umgesetzt. Bei den CPU-Zyklus-Messungen wurden sechs von acht Szenarien berücksichtigt.

Durch die Messreihen der Funktionen `gen_keypair`, `sign` und `verify` kann nicht festgestellt werden, ob es länger dauert eine Nachricht mit der Länge 128 Byte zu signieren als eine Nachricht der Länge 64 Byte. Dementsprechend wurde bei den Messreihen für die Szenarien angenommen, dass die Nachrichtenlänge bei der Signaturerstellung und -verifizierung nicht relevant ist.

Bei den Messungen der drei Funktionen ist Dilithium3 der Algorithmus, der bei allen drei Funktionen am wenigsten CPU-Zyklen benötigt. Die anderen Algorithmen variieren die Anzahlen der benötigten CPU-Zyklen für die Nachrichtengröße 512 Bit je nach Funktion. Dilithium5 ist an zweiter und Falcon-512 an dritter Stelle bei der Signaturerstellung, bei der Verifizierung tauschen die beiden die Pläze. Einige SPHINCS+-Parametersets sind bei der Schlüsselerstellung hingegen teilweise schneller als beide Falcon-Parametersets.

Bei den Messungen der CPU-Zyklen der einzelnen Szenarien fällt auf, dass Dilithium3 nicht bei allen Szenarien am wenigsten CPU-Zyklen benötigt, sondern verschiedene Algorithmen bei den unterschiedlichen Szenarien am schnellsten sind. Ansonsten wird festgestellt, dass die Szenarien, bei denen das Gerät Anfragen an den Server gestellt werden, mehr CPU-Zyklen benötigen. Bei den Szenarien 1, 2, 3b, 6, 8 sind Dilithium und Falcon am schnellsten. Bei den Szenarien 3a und 7 findet nur die erfolgreiche Verifizierung einer Signatur statt, sodass dabei nur eine der Hauptfunktionen durchgeführt wurde und die Algorithmen von SPHINCS+-SHAKE256-128s-simple sind in diesem Fall schneller als die der anderen beiden Algorithmen. Die anderen Algorithmen variieren in ihrer Performance bei den unterschiedlichen Szenarien. Bei erfolgreichem Verifizieren (Szenarien 3a und 7) sind die Algorithmen SPHINCS+-SHAKE256-128s-simple und SPHINCS+-SHAKE256-128s-robust schneller im Vergleich zu Falcon und Dilithium. Bei den allen anderen Szenarien benötigen die Dilithium- und Falcon-Parametersets weniger CPU-Zyklen als secp256r1. Die SPHINCS+- und RSA-Parametersets benötigen mehr CPU-Zyklen.

Eine weitere Erkenntnis ist, dass noch einige weitere Messungen durchgeführt werden sollten, damit die aus den durchgeführten Messreihen erhaltenen Ergebnisse besser erklärt werden können. Dazu wurden Vorschläge in Abschnitt 6.2 bei der Diskussion der Ergebnisse aufgeführt.

Unter den quantensicheren Algorithmen sind Falcon und Dilithium am schnellsten beim Durchführen eines Großteils der Szenarien. Des Weiteren ist Dilithium3 bei den Einzelmessungen der Funktionen am schnellsten. Die beiden Ergebnisse stellen Dilithium3 in Bezug auf die Performance als beste Wahl dar.

7.3. Beantwortung der Forschungsfragen

Die in Abschnitt 1.4.1 aufgeführten Forschungsziele wurden größtenteils erreicht (siehe Abschnitt 7.1). Die umgesetzten Ziele können im Abschnitt zu geleisteten Beiträgen (Abschnitt 7.1) und den einzelnen Antworten auf die Forschungsfragen entnommen werden. Eine Einschätzung über die Spezifikationen, die IoT-Geräte benötigen, um das Watchdog-Timer-Protokoll verwenden zu können, konnte nicht gegeben werden (siehe Abschnitt 7.3.5).

7.3.1. Forschungsfrage 1

Wie kann ein Watchdog-Timer-Protokoll unter Verwendung klassischer Kryptografie umgesetzt werden, um ein IoT-Gerät wiederherzustellen?

Um Forschungsfrage 1 zu beantworten, wurde im Konzeptionskapitel 3 dargestellt wie ein Watchdog-Timer-Protokoll theoretisch umgesetzt werden kann. Des Weiteren wurde in Kapitel 4 erläutert, dass das Protokoll mithilfe eines Aktorframeworks und der Bibliothek `python-mbedtls` in Python implementiert werden kann, um die Forschungsfrage zu beantworten. Mithilfe der Szenario-Messungen (Abschnitt 5.5.2) wurde gezeigt, dass die Umsetzung des Protokolls mit drei klassischen Algorithmen der Bibliothek `python-mbedtls` möglich ist.

Die genannte und getestete Proof-of-Concept-Implementierung ist nicht die einzige Möglichkeit, ein solches Protokoll umzusetzen. Weitere Möglichkeiten sind die für diese Arbeit als Grundlage genommenen Umsetzungen von Xu *et al.* [3] und Huber *et al.* [2].

7.3.2. Forschungsfrage 2

Wie kann ein Watchdog-Timer-Protokoll unter Verwendung quantensicherer Kryptografie umgesetzt werden, um ein IoT-Gerät wiederherzustellen?

Um Forschungsfrage 2 zu beantworten, wird die Antwort auf Frage 1 erweitert. Unter der Voraussetzung, dass das klassische Protokoll bereits umgesetzt wurde, können die `python-mbedtls`-Funktionen um die Funktionen für die Schlüsselerstellung, Signaturerstellung und Signaturverifizierung der `liboqs-python`-Bibliothek in der Implementierung (vgl. Kapitel 4) erweitert werden. Die Ersetzung erfolgt durch eine Funktion in der entschieden wird, ob die klassische oder quantensichere Bibliothek für die Schlüsselerstellung, Signaturerstellung und -verifizierung genutzt werden soll.

Die Ersetzung erfolgt durch eine weitere Auswahlmöglichkeit bei der Implementierung der Kryptografie. Die Wahl des zu verwendenden quantensicheren Signaturalgorithmus wird durch die `liboqs-python`-Bibliothek vereinfacht, sodass nur der Name des Algorithmus den Funktionen `gen_keypair`, `sign` und `verify` als Parameter übergeben werden muss. Mithilfe der Szenario-Messungen (Abschnitt 5.5.2) wurde gezeigt, dass die Umsetzung des Protokolls mit ausgewählten quantensicheren Algorithmen der Bibliothek `liboqs-python` möglich ist.

Des Weiteren wurde manuell von der Autorin die korrekte Reihenfolge und Durchführung der unterschiedlichen Szenarien geprüft.

7.3.3. Forschungsfrage 3

Wie hoch ist die Performance-Unterschied (in CPU-Zyklen) zwischen den Funktionen ausgewählter PQC-Algorithmen im Vergleich zu den Funktionen ausgewählter klassischer Algorithmen unter der Verwendung von Python-Wrappern?

In Abschnitt 6.2.2 werden die CPU-Zyklen der Funktionen ausgewählter PQC-Algorithmen mit den CPU-Zyklen der Funktionen ausgewählter klassischer Verfahren verglichen. Aus diesem Vergleich geht hervor, dass der Algorithmus Dilithium3 für alle drei betrachteten Funktionen (`gen_keypair`, `sign` und `verify`) schneller ist als die betrachteten klassischen Verfahren. Falcon-512, Falcon-1024 und Dilithium5 benötigen neben Dilithum3 am wenigsten CPU-Zyklen bei Schlüsselerstellung und -verifizierung. Falcon ist bei der Schlüsselerstellung vergleichsweise langsam, da das konzipierte Watchdog-Timer-Protokoll allerdings die Schlüsselerstellung nicht im Protokoll selbst nutzt, ist die dafür benötigte Zeit eher zweitrangig.

Bei der Signaturverifizierung ist jedoch der klassische Algorithmus rsa2048 an Platz drei und somit schneller als die meisten PQC-Algorithmen. Es kann demnach keine generelle Aussage getroffen werden und es muss für jede Funktion und jeden Algorithmus einzeln überprüft werden, ob die Anzahl der CPU-Zyklen bei PQC-Algorithmen geringer ist als die bei einem bestimmten klassischen Algorithmus.

Da bei Lazarus [2] mit secp256r1 als Algorithmus gearbeitet wurde, ist im Vergleich dazu Dilithium3 bei der Ausführung der Funktionen schneller.

Ein direkter Vergleich ist ggf. mit Vorsicht zu betrachten, da die Funktionen unterschiedliche Datentypen als Parameter entgegennehmen (vgl. Abschnitt 6.2.1).

7.3.4. Forschungsfrage 4

Wie hoch ist der Performance-Unterschied (in CPU-Zyklen) zwischen einem quantensicheren Watchdog-Timer-Protokoll im Vergleich zu einem klassischen Watchdog-Timer-Protokoll für unterschiedliche Szenarien?

In Abschnitt 5.5.2 wird in Tabellen aufgeführt, wie viele CPU-Zyklen die verwendeten Algorithmen benötigten, um bestimmte Szenarien auszuführen. Daraus geht hervor, dass der schnellste Algorithmus je nach gemessenem Szenario variiert. Bei den Szenarien 1, 2, 3b, 6 sind Falcon und Dilithium schneller als die drei klassischen Verfahren, gefolgt von SPHINCS+. Bei Szenarien 3b bzw. 6 und 8 ist SPHINCS+-SHA256-128f-simple jedoch schneller als secp256r1 bzw. rsa4096. Bei den Szenarien 3a und 7 sind die klassischen Algorithmen schneller als ein Großteil der PQC-Algorithmen. Ausnahmen sind dabei SPHINCS+-SHAKE256-128s-robust und -simple.

Bei manchen Szenarien sind die Algorithmen mit einem hohen NIST-Sicherheitslevel performanter als die klassischen Algorithmen secp256r1, rsa2048 und rsa4096. Z.B. ist secp256r1 bei den Szenarien 3a und 7 im Vergleich zu vielen quantensicheren Algorithmen langsamer. Bei den anderen Szenarien sind secp256r1 und rsa2048 unter den schnellsten sechs Algorithmen, wobei die gestesteten Falcon- und Dilithium-Parametersets etwas schneller sind. Dies zeigt, dass sich die Performance klassischer und quantensicherer Algorithmen nicht stark voneinander unterscheidet und durch eine Ersetzung mit quantensicheren Algorithmen kein Performance-Nachteil entsteht.

Ein direkter Vergleich ist ggf. mit Vorsicht zu betrachten, da die Funktionen unterschiedliche Datentypen als Parameter entgegennehmen (vgl. Abschnitt 6.2.1).

7.3.5. Forschungsfrage 5

Welche Eigenschaften sollte ein IoT-Gerät besitzen, um das quantensichere Watchdog-Timer-Protokoll nutzen zu können?

Es kann nicht pauschal festgelegt werden, welche Eigenschaften ein IoT-Gerät haben sollte, um das in dieser Arbeit neu konzipierte quantensichere Watchdog-Timer-Protokoll nutzen zu können. Das Watchdog-Timer-Protokoll hat je nach verwendetem Algorithmus unterschiedliche Anforderungen an das IoT-Gerät.

Im Paper von Bürstinghaus-Steinbach *et al.* [22] wurde der Einsatz von SPHINCS+ in der Bibliothek mbed TLS evaluiert. Auch dort wird die Herausforderung in der Zeit, die für die Signaturerstellung benötigt wird und die Größe der Signatur gesehen. Die Aussage in Bezug auf die benötigten CPU-Zyklen zum Erstellen der Signatur wurde in der vorliegenden Arbeit durch die Funktions- und Szenario-Messungen gezeigt, die zum Teil viel länger brauchen als die Algorithmen Falcon und Dilithium.

Die Implementierung in dieser Arbeit wurde jedoch ausschließlich in Software getestet und somit dient kein IoT-Gerät als Referenz. In dieser Arbeit wurden die für Funktionsaufrufe (Abschnitt 5.4.2) und die Durchführung bestimmter Szenarien (Abschnitt 5.5.2) benötigten CPU-Zyklen bestimmt. Um weitere Aussagen über die Verwendbarkeit des Watchdog-Timer-Protokolls unter Verwendung verschiedener quantensicherer Signaturalgorithmen im IoT-Bereich zu treffen, sollte auch der tatsächliche RAM-Verbrauch der Funktionen und Szenarien unter Verwendung der verschiedenen Algorithmen bestimmt und ausgewertet werden. Dazu können die Messfunktionen in der Datei `settings.py` ausgetauscht werden. Ein weiterer zu betrachtender Aspekt ist die Bestimmung der Größen der zu versendenden und empfangenden Nachrichten auch am Beispiel von entsprechend großen Updates.

Aus den Tabellen 2.3 und 3.2 geht hervor, dass Falcon-512 die kleinsten Signaturen (666 Byte) und SPHINCS+ die kleinsten öffentlichen (32, 48 bzw. 64 Byte) und privaten Schlüssel (64, 96 bzw. 128 Byte) erstellt. Aus den Funktionsmessungen geht allerdings hervor, dass Dilithium3 die beste Performance hat. Beim implementierten Protokoll schneiden jedoch für jedes Szenario unterschiedliche Algorithmen besser ab. In den meisten Szenarien benötigen die Falcon- und Dilithium-Algorithmen am wenigsten CPU-Zyklen.

Die Tendenz zur Wahl eines Algorithmus in Bezug auf die Performanz liegt bei Dilithium3 gefolgt, auch Dilithium5 und die beiden Falcon-Parametersets sind weit oben angesiedelt. Die Schlüsselerstellung unter Verwendung von Falcon-512 dauert jedoch 76 Mal länger als bei Dilithium3, bei Falcon-1024 sogar 219 Mal länger.

Es muss jeweils für den speziellen Anwendungsfall evaluiert werden, mit welchem Sicherheitslevel das Watchdog-Timer-Protokoll implementiert werden soll. Des Weiteren stellt sich die Frage, ob das IoT-Gerät vom Anwendungsfall bereits festgelegt wurde oder ob dieses auf Basis der Eigenschaften des Algorithmus festgelegt werden kann.

7.4. Ausblick auf zukünftige Forschung

In der Diskussion (Kapitel 6) wurden bereits einige Vorschläge für weiterführende Forschung gemacht, die in diesem Abschnitt kurz zusammengefasst werden.

Die Arbeit betrachtet die Technology Readiness Level nur bis einschließlich Stufe 3. Im Hinblick auf die Anwendung des Watchdog-Timer-Protokolls in realen Anwendungen kann es sinnvoll sein, die weiteren Schritte der TRL durchzuführen. Das Protokoll könnte auf zunächst zwei Hostsysteme - einen Server und ein Gerät - aufgeteilt werden und die Kommunikation somit über ein Netzwerk stattfinden. Später könnten weitere Server oder Geräte ergänzt werden, wobei die Implementierung in Bezug auf das Versenden der Nachrichten dann angepasst werden müsste. Im Zuge weiterer Schritte der TRL könnten die Vereinfachungen - wie die Updateversionen oder eine Sensor-API bei der Implementierung realer gestaltet werden. In weiterführender Forschung könnte außerdem das Protokoll um die Schlüsselerstellung und die -verteilung erweitert werden und in dem Zuge könnte auch die Speicherung der Schlüssel in quantensicheren Zertifikaten betrachtet werden.

Die Messungen der CPU-Zyklen wurden bisher nur auf einem Notebook durchgeführt. Um die Messergebnisse zu validieren, könnten die Messungen noch auf einem weiteren Rechner oder Server ausgeführt werden.Im Hinblick auf die Erklärung der Ergebnisse bei den Funktionsmessungen kann es sinnvoll sein, die aufgestellten Theorien mit den bereitgestellten Lösungsvorschlägen zu testen. Die Messungen der Funktionen könnten noch mit weiteren z.B. größeren Nachrichtenlängen durchgeführt werden, damit sich daraus ggf. die Abweichungen in der Performance von Dilithum3 bei den Szenarien im Vergleich zu den Funktionsmessungen erklären lassen. Bei den Messskripten der Funktionen wurden nur die CPU-Zyklen des erfolgreichen Verifizierens von Signaturen gemessen. In zukünftiger Forschung könnte vergleichsweise auch das Falsifizieren von Signaturen gebenchmarkt werden.

Bei diesen Messreihen könnten neben Dilithium2 auch noch weitere Signaturalgorithmen, die z.B. bei dem neuen Call for Proposals beim NIST eingereichten werden, oder weitere der genannten Python-Bibliotheken (Abschnitt 6.2.1), mit den bereits erstellten Messskripten getestet werden. Neben den CPU-Zyklen könnte noch der RAM-Verbrauch beim Ausführen der Funktionen oder Szenarien gemessen werden.

Zukünfte Forschung könnte an die Ergebnisse der Python-Benchmarks und die Benchmarks der C-Referenzimplementierungen anknüpfen und diese eingehender miteinander vergleichen und die Gründe der Abweichungen herausarbeiten.

Es wurde außerdem herausgefunden, dass die Implementierung mit einer zusätzlichen Hashfunktion und Überprüfung auf Gleichheit zweier Hashes teilweise mehr, teilweise aber auch weniger CPU-Zyklen benötigt, als wenn diese beiden Schritte nicht implementiert wird. In weiterführender Forschung könnten die Gründe dafür erforscht werden. Außerdem könnte evaluiert werden, welchen Einfluss ein zusätzliches Hashen - mit der gleichen oder einer anderen Hashfunktion - und ein zusätzlicher Vergleich auf die Sicherheit und die Performance des Protokolls unter Verwendung der unterschiedlichen Algorithmen hat.

Um an die Hardware-Implementierung des Lazarus-Prototyps anzuknüpfen, könnten dort quantensichere Algorithmen für die Signaturerstellung und Verifizierung eingesetzt werden. Alternative könnte das in dieser Arbeit konzipierte Watchdog-Timer-Protokoll in Hardware implementiert werden.

Eine weitere Alternative, die zu dem in dieser Arbeit konzipierten und implementierten Watchdog-Timer-Protokolls getestet werden kann, ist die Umsetzung der Implementierung mit dem Aktorframework *pykka*. Die Implementierungen und Benchmarks könnte anschließend miteinander verglichen werden.

Für jeden möglichen Anwendungsfall sollte evaluiert werden, welches IoT-Gerät für den Anwendungsfall in Kombination mit dem Watchdog-Timer-Protokoll infrage kommt, da jeder Anwendungsfall andere Anforderungen in Bezug auf das Gerät hat.

Um nicht nur auf die Sicherheit eines Algorithmus angewiesen zu sein, könnten die Implementierung des Watchdog-Timer-Protokolls hybrid umgesetzt werden. In diesem Fall heißt das, dass ein klassischer *und* ein quantensicherer Signaturalgorithmus implementiert wird.

A. Anhang

Im Anhang kann zunächst nachgelesen werden, wie die in dieser Arbeit beschriebene Proof-of-Concept-Implementierung installiert (Abschnitt A.1) und die verschiedenen Versionen genutzt (Abschnitt A.2) werden können. Anschließend wird der Inhalt der beiliegenden CD (Abschnitt A.3) sowie eine Gleichung zur Umrechnung von Cycles in Sekunden (Abschnitt A.4) aufgeführt. In Abschnitt A.5 werden alle Benchmarkings der Funktionsmessungen in Tabellenform dargestellt.

A.1. Installationsanleitung

Als Voraussetzung gelten folgende Programme (installiert via `apt`):

- Python 3.8
- python3.8-venv
- build-essential
- python3-dev

Des Weiteren kann das Proof-of-Concept nur auf einem Ubuntu-Rechner ausgeführt werden. Zuerst wird die Datei `Abgabe_MA_Gutsche.zip` heruntergeladen und entpackt.

Öffnen Sie im Ordner `Proof-of-Concept_QS_WDT` die Kommandozeile und erstellen eine virtuelle Umgebung für Python mit folgenden Kommandos:

```
cd Proof-of-Concept_QS_WDT
python3 -m venv .venv
source .venv/bin/activate
```

Sie sind nun in der virtuellen Python-Umgebung.

Mit `python3 --version` können Sie herausfinden, welche Python-Version in der Umgebung installiert ist.

Sollten Sie die Programme des Repositories später noch einmal ausführen wollen, müssen Sie sicherstellen, dass Sie mit

```
cd Proof-of-Concept_QS_WDT
source .venv/bin/activate
```

wieder die virtuelle Umgebung herstellen.

In dieser Umgebung installieren Sie noch die Anforderungen der Textdatei `requirements.txt`.

```
pip install -r requirements.txt
```

Um zu überpüfen, welche Packages in der virtuellen Umgebung vorhanden sind, führen Sie bitte folgendes Kommando aus:

```
pip list --local
```

Zusätzlich installieren sich noch weitere Packages, die die Abhängigkeiten von z.B. `numpy` sind.

A.1.0.1. Installation von liboqs und liboqs-python

Anschließend installiere die Abhängigkeiten für liboqs:

```
sudo apt install astyle cmake gcc ninja-build libssl-dev python3-pytest
python3-pytest-xdist unzip xsltproc doxygen graphviz python3-yaml valgrind
```

Anschließend entpacken Sie die Archive mit:

```
unzip liboqs.zip
unzip liboqs-python.zip
```

Um `liboqs-python` zu installieren, muss zunächst `liboqs` installiert werden unter Berücksichtigung der shared libs.

Gehe anschließend in den liboqs-Ordner, erstelle den build-Ordner und installiere die Bibliothek:

```
cd liboqs
```

```
mkdir build && cd build
cmake -GNinja .. -DBUILD_SHARED_LIBS=ON
ninja
```

```
sudo ninja install
```

Um die Bibliothek als shared-library nutzen zu können, führe aus:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
```

Gehe nun in die virtuelle Python-Umgebung und gehe in liboqs-python, führe aus:

```
cd ../../liboqs-python
python3 setup.py install
```

Danach muss der Pfad des aktuellen Ordners bestimmt werden:

```
pwd
> /home/tg/Abgabe_MA_Gutsche/Proof-of-Concept_QS_WDT/liboqs-python
```

Dieser wird daraufhin exportiert und dem PYTHONPATH hinzugefügt. Hinweis: Der Pfad muss entsprechend angepasst werden.

```
export PYTHONPATH=/home/tg/Abgabe_MA_Gutsche/Proof-of-Concept_QS_WDT/liboqs-python
```

A.1.0.2. Durchführung von CPU-Zyklus-Messungen

Überlegen Sie sich zunächst einen Algorithmus, mit dem Sie die CPU-Zyklus-Messungen durchführen wollen. Anschließend wechseln Sie in den Ordner `benchmarking` und führen das gewünschte Messskript mit den gewünschten Parametern (inkl. Algorithmus) für die Durchführung der Messung aus.

```
cd ../benchmarking
```

Um die CPU-Zyklen der liboqs-Funktionen zu messen, kann folgendes ausgeführt werden:

```
python3 liboqs_fkt_messskript.py --number=100 --variant=Falcon-512
--hash=sha512
```

Um die CPU-Zyklen der mbedtls-Funktionen zu messen, kann folgendes ausgeführt werden:

```
python3 mbedtls_fkt_messskript.py --number=100 --variant=secp256r1
--hash=sha512
```

Um die CPU-Zyklen eines bestimmten Szenarios des Protokolls zu messen, kann Folgendes ausgeführt werden:

```
cd ..
python3 messskript_szenarien.py --number=100 --crypto=pqc
--variant=Falcon-512 --scenario=1
```

Es gibt 8 verschiedene Szenarien, von denen 6 gemessen werden können.

A.1.0.3. Durchführung der Auswerteskripte

Die Auswerteskripte sind unabhängig von der Proof-of-Concept-Implementierung und den Messskripten. Daher muss im Ordner `Auswertung` erneut eine virtuelle Umgebung erstellt werden, in der die Auswerteskripte ausgeführt werden können.

```
cd Auswertung
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

Die .csv-Dateien der Messreihen werden in den Ordnern `benchmarking/functions` bzw. `benchmarking/scenarios` gespeichert und müssen im Ordner `Auswertung/Funktionen` bzw. `Auswertung/Szenarien` in die jeweiligen Ordner des NIST-Levels bzw. Szenarios kopiert werden und können dann darin ausgeführt werden. Dabei werden Auswertedateien im .csv-Format erstellt.

- Funktionen: `Klassisch`, `NIST_Level_1`, `NIST_Level_3`, `NIST_Level_5`, `Signer_Details`
- Szenarien: `Klassisch`, `NIST_Level_1`, `NIST_Level_3`, `NIST_Level_5` und darunter in eins der Szenarien: `1`, `2`, `3`, `6`, `7`, `8`

Die Auswertung der Funktionen erfolgt durch:

```
python3 Auswerteskript_Funktionen.py
```

Die Auswertung der Szenarien erfolgt durch:

```
python3 Auswerteskript_Szenarien.py
```

Die Auswertungen befinden sich im jeweils neben den Messungen mit der Benennung `Auswertung_*.csv`.

A.2. Nutzung der Versionen der Implementierung

Es gibt drei verschiedene Versionen der Implementierung, die auch auf der CD enthalten sind:

- Protokoll mit Messen ohne Hashing: im Ordner `Proof-of-Concept_QS_WDT`
- Protokoll ohne Messen ohne Hashing: im Ordner `Ohne_Messungen_ohne_Hashen`
- Messen mit Hashing: im Ordner `Messungen_mit_Hashing`

Die aktuelle Variante der Implementierung liegt im Ordner `Proof-of-Concept_QS_WDT` und ist für die Durchführung von Messungen vorgesehen. Bei dieser Version wird keine zusätzliche Hashfunktion vor den Funktionen `sign` und `verify` und keine manuelle Überprüfung von Hashes durchgeführt.

Um die Implementierung des Protokolls ohne die Einschränkungen der Vorkehrungen zum Messen der Szenarien zu erhalten, wird die Version des Codes (`Proof-of-Concept_QS_WDT.zip`) aus dem Ordner `Ohne_Messungen_ohne_Hashen` entpackt und mithilfe der Anleitung in Anhang A.1 installiert.

Zur Durchführung der Szenario-Messungen mit vorherigem Hashen und einer manuellen Überprüfung der Hashes, wird die Version des Codes (`Proof-of-Concept_QS_WDT.zip`) aus dem Ordner `Messungen_mit_Hashen` entpackt und mithilfe der Anleitung in Anhang A.1 installiert.

Um die Messungen mit denselben Startzuständen (Dateien und Schlüsseln) der zwei Szenario-Messungen erneut durchführen zu können, muss entweder der Inhalt des Ordners `temp_mit_Hashing` oder `temp_ohne_Hashing` in den `temp`-Ordner der jeweiligen Version `Proof-of-Concept_QS_WDT` ergänzt. Anschließend kann die jeweilige Version der Implementierung auf die jeweils benötigten Dateien und Schlüssel zugreifen.

A.3. Inhalte der CD

Auf der CD befinden sich folgende Ordner und Dateien:

- `Masterarbeit_Gutsche.pdf`
- `Alternative_Versionen_von_Teilen_des_Codes`
  - `Messungen_mit_Hashing`
    - `temp_mit_Hashing` ... Startzustände aller Algorithmen, die mit zusätzlichem Hashing erstellt wurden
    - `Proof-of-Concept_QS_WDT.zip` ..... Code mit zusätzlichem Hashing
  - `Messungen_ohne_Hashing`
    - `temp_ohne_Hashing` .. Startzustände aller Algorithmen, die ohne zusätzliches Hashing erstellt wurden
  - `Ohne_Messungen_ohne_Hashing`
    - `Proof-of-Concept_QS_WDT.zip` ..... Code ohne zusätzliches Hashing
- `Auswertung`
  - `Auswertungen_MA`
    - `Funktionen_gemessen` ..... Auswertungen der gemessenen Funktionen
    - `Szenarien_mit_Hash` ..... Auswertungen der gemessenen Szenarien mit zusätzlichem Hashing
    - `Szenarien_ohne_Hashing` .... Auswertungen der gemessenen Szenarien ohne zusätzliches Hashing
  - `Funktionen` ..... leerer Ordner zum neu Befüllen
  - `Szenarien` ..... leerer Ordner zum neu Befüllen

```
    ├── Auswerteskript_Funktionen.py
    ├── Auswerteskript_Szenarien.py
    └── requirements.txt ..... Für Auswerteskripte
Proof-of-Concept_QS_WDT
    ├── .vscode
    ├── benchmarking
    │   ├── measurements
    │   │   ├── functions
    │   │   └── scenarios
    │   ├── memory
    │   │   ├── device_secure_storage
    │   │   ├── device_staging_area
    │   │   ├── server_data_storage
    │   │   └── server_secure_storage
    │   ├── liboqs_fkt_messskript.py
    │   └── mbedtls_fkt_messskript.py
    ├── measurements
    │   └── functions
    ├── liboqs ..... Bibliothek für PQC-Algorithmen
    ├── liboqs-python ..... Python-Wrapper-Bibliothek für liboqs
    ├── memory ..... Enthält vier Ordner zum Speichern siehe memory weiter oben
    ├── modules
    │   ├── common.py
    │   ├── crypto.py
    │   ├── filehandling.py
    │   ├── generate_objects.py
    │   ├── keys.py
    │   ├── messagetypes.py
    │   ├── remove.py
    │   └── utils.py
    ├── temp ..... gespeicherte Dateien und Schlüssel für Startzustände
    ├── app.py
    ├── device.py
    ├── key_generation.py
    ├── messskript_szenarien.py
    ├── Proof-of-Concept_QS_WDT.code-workspace
    ├── README.md
    ├── requirements.txt ..... Für Implementierung und Messskripte
    ├── scenario_settings.py
    ├── server.py
    └── settings.py
```

A.4. Umrechnung von Cycles in Sekunden

Die Messwerte der durchgeführten Messreihen liegen ausschließlich in CPU-Zyklen (Cycles) vor. Bei Bedarf können die Messwerte in Cycles unter Zuhilfenahme der Taktfrequenz des für die Messungen verwendeten Prozessors in Sekunden umgerechnet werden.

Das in dieser Arbeit verwendete Notebook hat eine CPU-Taktfrequenz von 2,60 GHz (vgl. 5.1.1). Das bedeutet, dass die CPU pro Sekunde 2,6 Milliarden Takte bzw. Zyklen durchführen kann. Somit wird Gleichung A.1 zur Umrechnung von Zyklen in Sekunden verwendet.

$$1/(2,60 * 10^9) * Messwert_{Cycles} = Messwert_{Sekunden} \tag{A.1}$$

A.5. Benchmarkings für die Funktionen

In diesem Abschnitt werden die ausgewerteten Messreihen der CPU-Zyklen für die Funktionen `gen_keypair`, `sign` und `verify` für verschiedene Algorithmen aus ggf. unterschiedlichen Bibliotheken aufgeführt. Die Tabellen sind aufgeteilt in die Kategorien: klassisch, NIST Level 1, NIST Level 3 und NIST Level 5.

Bei allen Messungen zur Signaturerstellung und -verifizierung wurden Nachrichten unterschiedlicher Länge signiert bzw. verifiziert. Bei der Verwendung von der Hashfunktion sha256 beim Signieren wurde eine Nachricht der Länge 64 Byte signiert. Wurde sha384 bzw sha512 als Hashfunktion beim Signieren verwendet, so wurde eine Nachricht der Länge 96 Byte bzw. 128 Byte signiert. Gleiches gilt für das Verifizieren.

A.5.1. Schlüsselerstellung mit `gen_keypair`

In diesem Abschnitt sind die Auswertungen für die CPU-Zyklen bei der Schlüsselerstellung unter Verwendung klassischer und quantensicherer Verfahren aufgeteilt in klassisch (Tabelle A.1), Level 1 (Tabelle A.2), Level 3 (Tabelle A.3) und Level 5 (Tabelle A.4) der NIST-Anforderungen.

Bei den Messungen zur Schlüsselerstellung könnten alle drei Messreihen unter Verwendung der unterschiedlichen Hashfunktionen als eine Messreihe angesehen werden, da die Schlüsselerstellung unabhängig von der beim Signieren bzw. Verifizieren verwendeten Hashfunktion ist.

Algorithmus	Hash	Mittelwert (Cycles)	Median (Cycles)	Standard-abweichung (Cycles)	Minimaler Wert (Cycles)	Maximaler Wert (Cycles)
rsa2048	sha256	393.633.605	339.410.880	246.962.729	38.076.924	2.153.514.621
	sha384	394.109.396	337.892.716	249.681.188	37.869.687	2.518.762.352
	sha512	397.112.059	336.380.018	254.907.473	40.859.256	1.971.242.734
rsa4096	sha256	4.288.661.454	3.573.293.709	2.978.785.248	189.417.926	21.439.498.507
	sha256_2	4.233.773.422	3.555.034.220	2.964.628.068	189.885.369	25.109.411.320
	sha384	4.251.413.364	3.589.151.463	2.919.594.717	158.886.829	24.730.514.207
	sha512	4.254.680.764	3.603.927.232	2.908.547.659	212.638.835	24.512.202.849
secp256r1	sha256	3.537.081	3.306.422	639.915	3.149.923	11.758.133
	sha384	3.525.281	3.282.320	639.073	3.140.444	12.040.915
	sha512	3.534.409	3.299.562	639.972	3.146.817	13.037.075

Tabelle A.1.: Auswertung der CPU-Zyklen der Messreihen zur klassischen Schlüsselerstellung unter Verwendung der Bibliothek `python-mbedtls`.

Algorithmus		Hash	Mittelwert (Cycles)	Median (Cycles)	Standard-abweichung (Cycles)	Minimaler Wert (Cycles)	Maximaler Wert (Cycles)
Falcon-512		sha256	22.864.020	21.133.944	6.010.655	16.195.251	65.106.834
		sha384	22.737.999	20.973.285	6.057.079	16.204.572	73.167.433
		sha512	22.759.020	21.045.430	5.907.319	16.203.509	61.710.902
SPHINCS+-SHA256	128f-robust	sha256	3.249.714	2.964.915	702.913	2.789.867	12.744.208
		sha384	3.264.797	2.966.206	735.600	2.787.139	11.781.192
		sha512	3.262.297	2.975.035	716.795	2.787.383	12.385.411
	128f-simple	sha256	1.771.009	1.630.462	356.844	1.523.283	9.109.750
		sha384	1.757.693	1.626.880	321.438	1.533.876	8.741.889
		sha512	1.781.790	1.624.763	403.366	1.509.325	7.940.190
	128s-robust	sha256	184.294.036	182.218.821	11.259.569	172.334.552	334.498.342
		sha384	185.387.379	183.240.652	12.435.531	172.775.964	341.174.368
		sha512	184.296.598	182.899.016	9.503.367	173.130.667	332.352.046
	128s-simple	sha512	100.174.991	99.538.284	3.590.260	93.910.907	175.697.993
SPHINCS+-SHAKE256	128f-robust	sha256	7.316.182	6.965.646	1.531.881	5.689.188	24.503.974
		sha384	7.373.281	6.957.804	1.620.787	5.697.127	22.544.440
		sha512	7.428.818	6.950.860	1.741.826	5.710.211	23.153.880
	128f-simple	sha256	4.028.101	3.712.336	982.609	3.300.255	16.009.699
		sha384	4.060.226	3.716.880	1.039.143	3.309.604	15.405.781
		sha512	4.021.943	3.693.146	1.003.958	3.317.588	18.352.276
	128s-robust	sha512	419.178.911	414.458.092	27.727.627	371.028.338	607.514.020
	128s-simple	sha512	230.808.864	227.203.906	19.495.229	20.4518.917	396.689.864

Tabelle A.2.: Auswertung der CPU-Zyklen der Messreihen zur quantensicheren Schlüsselerstellung mit NIST-Level 1 unter Verwendung der Bibliothek `liboqs-python`.

Algorithmus		Hash	Mittelwert (Cycles)	Median (Cycles)	Standard-abweichung (Cycles)	Minimaler Wert (Cycles)	Maximaler Wert (Cycles)
Dilithium3		sha256	300.407	254.538	133.373	203.501	1.817.273
		sha384	293.885	245.006	128.660	203.249	2.023.792
		sha512	300.743	251.088	139.997	204.018	1.776.677
SPHINCS+-SHA256	192f-robust	sha256	4.471.980	4.196.575	926.163	4.049.927	18.951.788
		sha384	4.515.731	4.197.051	1.010.981	4.046.472	16.725.014
		sha512	4.535.912	4.192.117	1.127.443	4.040.212	19.946.493
	192f-simple	sha256	2.563.663	2.387.994	492.451	2.264.040	9.758.934
		sha384	2.573.409	2.388.308	510.847	2.252.802	8.897.278
		sha512	2.579.888	2.384.791	542.706	2.238.848	9.528.491
	192s-robust	sha256	267.342.600	263.775.534	9.096.947	256.815.447	449.665.673
		sha384	270.348.819	265.156.898	18.857.049	257.374.495	437.566.325
		sha512	267.825.728	264.051.200	13.628.456	257.057.724	430.096.141
	192s-simple	sha512	147.357.530	144.962.498	7.576.394	140.684.807	285.001.542
SPHINCS+-SHAKE256	192f-robust	sha256	10.326.528	9.816.452	1.705.880	8.441.200	35.387.966
		sha384	10.376.525	9.866.046	1.569.824	8.370.401	23.742.831
		sha512	10.266.427	9.845.368	1.503.168	8.347.741	30.784.720
	192f-simple	sha256	5.758.583	5.301.878	1.404.427	4.849.417	19.994.156
		sha384	5.797.196	5.314.613	1.451.748	4.662.574	22.527.069
		sha512	5.817.466	5.304.622	1.468.967	4.868.148	19.089.640
	192s-robust	sha512	593.878.619	588.303.072	30.480.024	545.742.338	844.180.308
	192s-simple	sha512	342.672.174	336.955.478	27.017.275	303.801.712	531.540.618

Tabelle A.3.: Auswertung der CPU-Zyklen der Messreihen zur quantensicheren Schlüsselerstellung mit NIST-Level 3 unter Verwendung der Bibliothek `liboqs-python`.

Algorithmus		Hash	Mittelwert (Cycles)	Median (Cycles)	Standard-abweichung (Cycles)	Minimaler Wert (Cycles)	Maximaler Wert (Cycles)
Dilithium5		sha256	411.712	355.418	151.773	291.681	1.801.768
		sha384	412.718	356.001	156.326	293.654	2.452.935
		sha512	410.747	354.211	151.422	294.796	2.607.259
Falcon-1024		sha256	65.965.421	60.533.830	16.349.004	50.148.226	199.417.134
		sha384	66.200.178	60.838.198	16.807.310	50.220.043	218.767.140
		sha512	65.758.137	60.339.460	16.354.462	50.105.808	227.175.155
SPHINCS+-SHA256	256f-robust	sha256	21.218.569	20.997.740	1.826.367	18.922.870	61.247.218
		sha384	21.242.306	21.024.246	1.672.906	18.931.711	56.522.516
		sha512	21.349.798	21.098.042	1.942.411	18.948.583	71.254.443
	256f-simple	sha256	6.329.413	5.700.998	1.318.063	5.455.727	20.169.893
		sha384	6.328.283	5.690.136	1.349.226	5.463.672	25.559.475
		sha512	6.282.131	5.698.364	1.253.660	5.457.794	26.100.961
	256s-robust	sha512	322.962.530	317.633.744	21.432.275	305.937.196	493.955.930
	256s-simple	sha512	93.143.807	91.929.185	5.986.577	86.583.979	203.974.574
SPHINCS+-SHAKE256	256f-robust	sha256	26.162.237	25.451.238	3.069.938	22.157.241	85.611.760
		sha384	26.099.690	25.358.136	3.081.103	22.388.740	85.380.811
		sha512	26.395.655	25.593.006	3.615.928	22.369.860	83.988.066
	256f-simple	sha256	14.962.007	14.464.826	2.205.340	12.627.296	47.438.268
		sha384	14.951.014	14.474.520	2.217.634	12.468.519	51.262.852
		sha512	14.916.712	14.466.245	2.057.918	12.727.949	41.998.512
	256s-robust	sha512	397.752.949	391.848.819	27.015.170	364.096.385	642.789.455
	256s-simple	sha512	230.422.921	226.330.976	20.301.093	202.514.668	401.485.873

Tabelle A.4.: Auswertung der CPU-Zyklen der Messreihen zur quantensicheren Schlüsselerstellung mit NIST-Level 5 unter Verwendung der Bibliothek `liboqs-python`.

A.5.2. Signaturerstellung mit `sign`

In diesem Abschnitt sind die Auswertungen für die CPU-Zyklen bei der Signaturerstellung unter Verwendung klassischer und quantensicherer Verfahren aufgeteilt in klassisch (Tabelle A.5), Level 1 (Tabelle A.6), Level 3 (Tabelle A.7) und Level 5 (Tabelle A.8) der NIST-Anforderungen.

Algorithmus	Hash	Mittelwert (Cycles)	Median (Cycles)	Standard-abweichung (Cycles)	Minimaler Wert (Cycles)	Maximaler Wert (Cycles)
rsa2048	sha256	10.994.658	10.660.416	1.233.149	10.358.114	2.1558.620
	sha384	10.992.283	10.663.952	1.228.033	10.347.549	30.204.077
	sha512	11.054.261	10.722.024	1.262.677	10.379.275	35.728.107
rsa4096	sha256	54.938.864	54.474.689	2.358.949	51.270.030	72.061.557
	sha256_2	54.649.228	54.295.774	2.098.959	51.244.437	7.487.3350
	sha384	54.855.727	54.396.712	2.210.500	51.098.482	77.254.317
	sha512	54.454.846	54.049.258	2.211.346	51.058.228	111.463.635
secp256r1	sha256	3.825.267	3.565.154	695.627	3.367.248	11.689.714
	sha384	3.833.641	3.548.200	726.030	3.368.257	15.740.219
	sha512	3.812.381	3.559.204	662.553	3.373.693	12.402.504

Tabelle A.5.: Auswertung der CPU-Zyklen der Messreihen zur klassischen Signaturerstellung unter Verwendung von `python-mbedtls`.

Algorithmus		Hash	Mittelwert (Cycles)	Median (Cycles)	Standard-abweichung (Cycles)	Minimaler Wert (Cycles)	Maximaler Wert (Cycles)
Falcon-512		sha256	931.455	832.240	235.690	758.172	4.681.508
		sha384	964.440	856.600	251.776	782.354	3.511.221
		sha512	923.573	832.906	219.448	762.288	4.535.037
SPHINCS+-SHA256	128f-robust	sha256	81.590.744	79.528.184	6.481.058	75.650.105	185.168.643
		sha384	81.950.661	79.577.004	7.533.848	75.696.228	192.025.782
		sha512	82.119.065	79.606.270	7.294.536	75.971.216	189.787.193
	128f-simple	sha256	43.457.524	43.087.426	2.965.901	40.051.131	102.804.812
		sha384	43.375.726	43.140.800	2.685.115	40.146.805	75.015.950
		sha512	43.789.247	43.105.640	4.381.893	39.899.754	115.150.594
	128s-robust	sha256	1.343.492.719	1.336.675.905	31.908.416	1.291.224.307	1.550.078.738
		sha384	1.367.981.007	1.361.726.594	40.911.407	1.304.475.166	1.554.359.240
		sha512	1.348.971.277	1.333.158.288	43.570.216	1.291.317.075	2.303.945.216
	128s-simple	sha512	718.696.223	716.276.206	13.709.138	707.833.195	1.284.484.789
SPHINCS+-SHAKE256	128f-robust	sha256	178.524.180	174.936.500	14.525.786	160.662.458	314.319.876
		sha384	176.165.147	174.628.726	89.105.27	158.329.782	30.352.0799
		sha512	179.451.457	175.070.128	17.629.180	157.457.453	318.189.666
	128f-simple	sha256	104.113.111	102.091.246	8.529.783	92.113.150	217.953.623
		sha384	104.175.348	102.020.717	9.050.370	92.126.103	208.245.455
		sha512	103.962.746	101.816.930	8.562.325	92.854.465	199.741.175
	128s-robust	sha512	3.048.306.126	3.048.752.912	95.202.676	2.710.401.307	3.879.631.533
	128s-simple	sha512	1.673.651.683	1.673.429.308	60.388.393	1.516.060.940	2.044.074.603

Tabelle A.6.: Auswertung der CPU-Zyklen der Messreihen zur quantensicheren Signaturerstellung mit NIST-Level 1 unter Verwendung von `liboqs-python`.

Algorithmus		Hash	Mittelwert (Cycles)	Median (Cycles)	Standard-abweichung (Cycles)	Minimaler Wert (Cycles)	Maximaler Wert (Cycles)
Dilithium3		sha256	527.255	437.028	305.286	230.094	3.535.783
		sha384	529.258	435.745	317.616	228.636	3.306.959
		sha512	534.011	441.538	317.155	229.369	3.116.232
SPHINCS+-SHA256	192f-robust	sha256	131.771.561	129.205.170	8.536.707	123.702.048	260.675.972
		sha384	131.980.803	129.241.600	9.030.225	123.804.223	257.883.531
		sha512	132.396.750	129.221.969	10.895.801	123.616.517	261.736.964
	192f-simple	sha256	73.351.990	71.727.730	5.557.382	68.630.877	164.733.547
		sha384	73.587.548	71.818.242	6.071.491	68.556.228	173.645.233
		sha512	73.984.259	71.846.222	7.053.867	68.692.023	165.714.359
	192s-robust	sha256	2.490.327.606	2.481.127.057	41.806.126	2.455.869.067	3.344.357.405
		sha384	2.504.564.189	2.505.701.722	39.224.422	2.454.679.821	3.028.135.376
		sha512	2.492.905.109	2.488.957.632	32.880.047	2.454.358.567	2.852.179.570
	192s-simple	sha512	1.393.320.032	1.390.218.108	23.781.124	1.367.745.219	1.688.557.699
SPHINCS+-SHAKE256	192f-robust	sha256	278.321.527	272.792.460	23.393.643	242.995.428	440.251.706
		sha384	274.460.247	271.111.191	17.813.538	246.691.444	464.416.777
		sha512	275.719.036	269639.836	24.177.370	242.681.128	438.304.750
	192f-simple	sha256	160.743.270	156.995.063	14.132.939	144.386.445	293.692.890
		sha384	161.644.152	157.212.957	15.755.272	143.417.941	298.439.459
		sha512	161.507.971	157.435.028	14.873.700	142.112.926	288.680.247
	192s-robust	sha512	5.197.246.435	5.212.681.626	122.221.235	4.690.116.051	5.823.069.318
	192s-simple	sha512	3.076.841.312	3.074.878.266	76.031.847	2.825.506.357	3699.227.893

Tabelle A.7.: Auswertung der CPU-Zyklen der Messreihen zur quantensicheren Signaturerstellung mit NIST-Level 3 unter Verwendung von `liboqs-python`.

Algorithmus		Hash	Mittelwert (Cycles)	Median (Cycles)	Standard-abweichung (Cycles)	Minimaler Wert (Cycles)	Maximaler Wert (Cycles)
Dilithium5		sha256	630.292	542.183	314.172	330.902	5.674.588
		sha384	643.736	546.504	327.130	332.557	3.240.809
		sha512	643.154	545.316	337.723	332.294	5.463.191
Falcon-1024		sha256	1.730.819	1.594.514	360.113	1.483.179	8.245.207
		sha384	1.751.617	1.592.150	401.590	1.479.487	6.701.908
		sha512	1.800.617	1.643.014	405.184	1.533.495	7.036.597
SPHINCS+-SHA256	256f-robust	sha256	418.124.768	412.413.804	23.453.416	398.700.331	605.686.605
		sha384	419.695.467	413.179.962	25.918.251	399.020.504	705.388.563
		sha512	421.200.951	414.557.173	27.133.107	399.228.353	665.385.587
	256f-simple	sha256	133.573.666	130.229.632	10.587.047	124.309.989	265.658.098
		sha384	133.710.751	130.134.562	11.420.434	124.311.700	264.061.374
		sha512	131.994.956	129.631.940	7.718.424	124.098.899	250.521.700
	256s-robust	sha512	3.612.937.638	3.623.840.965	59.015.706	3.524.842.031	3.982.222.365
	256s-simple	sha512	1.114.506.228	1.110.573.992	28.316.220	1.084.621.876	1.336.010.237
SPHINCS+-SHAKE256	256f-robust	sha256	521.308.947	513.878.740	32.012.652	471.304.727	711.868.309
		sha384	526.586.503	518.457.303	32.932.344	473.419.469	777.168.419
		sha512	527.580.182	518.345.416	37.848.266	473.263.934	898.741.850
	256f-simple	sha256	300.974.045	295.444.656	22.994.687	273.239.461	474.225.651
		sha384	303.783.301	297.661.960	24.946.180	275.107.678	473.927.106
		sha512	301.455.891	296.718.966	21.552.488	271.264.172	483.238.917
	256s-robust	sha512	4.467.468.887	4.477.511.515	94.995.020	4.019.125.716	5.329.922.520
	256s-simple	sha512	2.653.550.216	2.649.342.503	66.017.279	2.458.736.956	3.173.449.961

Tabelle A.8.: Auswertung der CPU-Zyklen der Messreihen zur quantensicheren Signaturerstellung mit NIST-Level 5 unter Verwendung von `liboqs-python`.

A.5.3. Signaturverifizierung mit `verify`

In diesem Abschnitt sind die Auswertungen für die CPU-Zyklen bei der Signaturverifizierung unter Verwendung klassischer und quantensicherer Verfahren aufgeteilt in klassisch (Tabelle A.9), Level 1 (Tabelle A.10), Level 3 (Tabelle A.11) und Level 5 (Tabelle A.12) der NIST-Anforderungen.

Algorithmus	Hash	Mittelwert (Cycles)	Median (Cycles)	Standard-abweichung (Cycles)	Minimaler Wert (Cycles)	Maximaler Wert (Cycles)
rsa2048	sha256	277.092	263.822	58.507	245.530	1.128.139
	sha384	277.196	263.838	62.591	247.959	1.148.009
	sha512	279.735	265.846	64.273	248.759	1.418.671
rsa4096	sha256	852.484	795.944	164.018	729.465	3.131.796
	sha256_2	852.538	793.800	165.405	738.027	3.230.376
	sha384	844.838	797.346	146.885	735.673	3.562.717
	sha512	848.945	791.302	156.943	728.996	3.333.437
secp256r1	sha256	7.241.686	6.863.701	1.004.834	6.557.579	18.542.053
	sha384	7.253.651	6.855.398	1.033.344	6.543.247	18.845.747
	sha512	7.214.049	6.869.426	962.627	6.560.369	19.428.410

Tabelle A.9.: Auswertung der CPU-Zyklen der Messreihen zur klassischen Signaturverifizierung unter Verwendung von `python-mbedtls`.

Algorithmus		Hash	Mittelwert (Cycles)	Median (Cycles)	Standard-abweichung (Cycles)	Minimaler Wert (Cycles)	Maximaler Wert (Cycles)
Falcon-512		sha256	244.754	216.029	99.448	161.971	1.762.776
		sha384	250.671	224.588	109.390	163.421	2.532.584
		sha512	244.521	217.842	97.969	163.249	2.086.513
SPHINCS+-SHA256	128f-robust	sha256	13.236.522	12.485.884	2.156.073	11.289.494	49.455.994
		sha384	13.412.280	12.569.642	2.333.611	11.288.441	50.551.190
		sha512	13.355.154	12.536.122	2.229.086	11.210.792	48.031.723
	128f-simple	sha256	6.852.675	6.489.437	1.119.342	5.775.224	21.054.778
		sha384	6.807.456	6.482.810	988.008	5.826.409	16.634.722
		sha512	6.902.159	6.509.127	1.236.570	5.804.506	31.901.492
	128s-robust	sha256	4.333.254	4.173.782	874.967	3.623.771	15.714.234
		sha384	4.400.044	4.179.489	983.024	3.614.806	15.487.891
		sha512	4.351.602	4.182.812	864.039	3.655.444	14.256.128
	128s-simple	sha512	2.236.852	2.204.986	299.844	1.903.472	9.024.593
SPHINCS+-SHAKE256	128f-robust	sha256	21.452.056	20.574.536	3.795.534	16.462.796	66.633.722
		sha384	21.245.237	20.438.424	3.250.649	16.370.715	71.979.380
		sha512	21.311.036	20.377.314	3.779.205	16.180.300	67.988.151
	128f-simple	sha256	11.499.328	10.974.148	1.909.750	8.481.036	36.132.662
		sha384	11.533.615	10.962.204	2.030.043	8.726.401	40.663.795
		sha512	11.613.421	11.075.992	1.931.198	8.693.822	41.295.332
	128s-robust	sha512	7.208.798	7.132.662	757454	5487211	20582221
	128s-simple	sha512	3.807.469	3.739.587	553153	2775011	14004953

Tabelle A.10.: Auswertung der CPU-Zyklen der Messreihen zur klassischen Signaturverifizierung mit NIST-Level 1 unter Verwendung von `liboqs-python`.

Algorithmus		Hash	Mittelwert (Cycles)	Median (Cycles)	Standard-abweichung (Cycles)	Minimaler Wert (Cycles)	Maximaler Wert (Cycles)
Dilithium3		sha256	205.047	180.708	58.285	169.902	1.444.812
		sha384	205.031	180.192	57.751	170.869	982.821
		sha512	208.293	180.862	67.509	171.684	1.419.565
SPHINCS+-SHA256	192f-robust	sha256	19.188.883	18.399.748	2.603.970	17.200.863	66.277.861
		sha384	19.338.625	18.409.068	3.000.964	17.250.107	72.586.930
		sha512	19.496.514	18.455.271	3.354.541	17.244.627	66.326.839
	192f-simple	sha256	10.239.267	9.715.046	1.576.226	8.917.920	38.629.441
		sha384	10.272.515	9.722.986	1.665.737	8.853.869	41.781.486
		sha512	10.314.519	9.742.314	1.744.747	9.009.309	42.252.502
	192s-robust	sha256	6.609.988	6.537.437	535.694	5.943.980	20.461.989
		sha384	6.610.146	6.565.574	437.651	5.943.199	15.233.801
		sha512	6.591.655	6.538.390	469.929	5.907.429	16.737.740
	192s-simple	sha512	3.415.077	3.368.828	381.200	3.018.320	12.758.502
SPHINCS+-SHAKE256	192f-robust	sha256	32.653.778	32.094.656	5.573.170	24.064.722	101.677.445
		sha384	32.247.851	31.791.224	5.118.822	24.085.781	102.091.119
		sha512	31.737.901	31.063.218	5.310.314	23.968.678	101.618.692
	192f-simple	sha256	16.769.810	16.076.853	3.027.596	12.406.131	52.753.761
		sha384	16.819.405	16.066.928	3.051.488	12.517.614	55.514.076
		sha512	16.703.478	15.929.610	3.150.544	12.424.502	54.514.898
	192s-robust	sha512	10.249.724	10.232.792	886276	8120401	32141341
	192s-simple	sha512	5.373.030	5.344.820	496.621	4.212.093	17.586.880

Tabelle A.11.: Auswertung der CPU-Zyklen der Messreihen zur klassischen Signaturverifizierung mit NIST-Level 3 unter Verwendung von `liboqs-python`.

Algorithmus		Hash	Mittelwert (Cycles)	Median (Cycles)	Standard-abweichung (Cycles)	Minimaler Wert (Cycles)	Maximaler Wert (Cycles)
Dilithium5		sha256	302.632	269.417	76.785	255.823	1.477.730
		sha384	305.898	270.014	82.563	255.852	1.697.984
		sha512	305.690	270.706	80.582	256.715	1.469.787
Falcon-1024		sha256	407.722	397.970	118.925	298.396	1.590.710
		sha384	411.969	400.638	130.821	298.847	1.884.025
		sha512	406.840	394.215	122.202	300.364	2.047.782
SPHINCS+-SHA256	256f-robust	sha256	24.420.408	23.966.582	2.488.958	22.499.371	84.443.075
		sha384	24.419.395	23.997.024	2.286.187	22.428.328	74.438.902
		sha512	24.546.816	24.043.316	2.609.326	22.456.307	85.728.414
	256f-simple	sha256	10.486.330	9.662.062	1.869.984	8.929.085	31.343.313
		sha384	10.506.263	9.668.441	1.957.164	8.886.491	42.873.960
		sha512	10.384.222	9.626.006	1.738.142	8.895.646	31.601.691
	256s-robust	sha512	12.322.329	12.256.458	788.708	11.229.059	34.769.343
	256s-simple	sha512	4.875.969	4.833.171	406.233	4.345.481	13.184.080
SPHINCS+-SHAKE256	256f-robust	sha256	32.670.057	32.168.690	5.044.323	24.309.859	99.518.021
		sha384	32.863.961	32.283.501	5.286.286	24.615.142	95.769.266
		sha512	30.334.118	29.145.753	4.463.508	24.645.891	91.644.410
	256f-simple	sha256	16.134.987	15.764.758	2.291.727	12.619.320	56.890.950
		sha384	16.175.840	15.772.146	2.276.244	12.645.602	54.150.873
		sha512	16.137.057	15.816.140	2.119.780	12.729.461	54.727.696
	256s-robust	sha512	14.801.720	14.823.982	1.213.167	11.932.068	48.007.891
	256s-simple	sha512	7.853.349	7.800.105	868.717	6.202.935	27.020.467

Tabelle A.12.: Auswertung der CPU-Zyklen der Messreihen zur klassischen Signaturverifizierung mit NIST-Level 5 unter Verwendung von `liboqs-python`.

Literaturverzeichnis

[1] B. Dorsemaine, J.-P. Gaulier, J.-P. Wary, N. Kheir, and P. Urien, "Internet of Things: A Definition & Taxonomy," in *2015 9th International Conference on Next Generation Mobile Applications, Services and Technologies*, pp. 72–77.

[2] M. Huber, S. Hristozov, S. Ott, V. Sarafov, and M. Peinado, "The Lazarus Effect: Healing Compromised Devices in the Internet of Small Things," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '20. Association for Computing Machinery, pp. 6–19, accessed on: 2022-05-04. [Online]. Available: https://doi.org/10.1145/3320269.3384723

[3] M. Xu, M. Huber, Z. Sun, P. England, M. Peinado, S. Lee, A. Marochko, D. Mattoon, R. Spiger, and S. Thom, "Dominance as a New Trusted Computing Primitive for the Internet of Things," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 1415–1430, accessed on: 2022-04-21. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8835213

[4] M. A. F. Al-Husainy, B. Al-Shargabi, and S. Aljawarneh, "Lightweight cryptography system for IoT devices using DNA," vol. 95, p. 107418, accessed on: 2022-06-30. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045790621003827

[5] M. Klein-Hennig and F. Schmidt, "Zurück auf los — die it-sicherheit zurück in der steinzeit: Haftung und lösungen für it-sicherheitsmängel im internet of things," vol. 41, no. 10, pp. 605–611, accessed on: 2022-06-29. [Online]. Available: http://link.springer.com/10.1007/s11623-017-0841-9

[6] M. Wilczek, "Iot – die unterschätzte gefahr für it-sicherheit," vol. 45, no. 2, pp. 79–82, accessed on: 2022-06-29. [Online]. Available: http://link.springer.com/10.1007/s11623-021-1394-5

[7] Bundesamt für Sicherheit in der Informationstechnik (BSI). Kryptografie quantensicher gestalten: Grundlagen, Entwicklungen, Empfehlungen. Accessed on: 2022-06-28. [Online]. Available: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Broschueren/Kryptografie-quantensicher-gestalten.pdf?__blob=publicationFile&v=4

[8] P. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pp. 124–134.

[9] A. Canteaut, S. Duval, G. Leurent, M. Naya-Plasencia, L. Perrin, T. Pornin,

and A. Schrottenloher, “Saturnin: A suite of lightweight symmetric algorithms for post-quantum security,” accessed on: 2022-05-12. [Online]. Available: https://hal.inria.fr/hal-02436763

[10] Bundeskriminalamt (BKA). BKA - Lageprodukte aus dem Bereich Cybercrime - Bundeslagebild Cybercrime 2021. Accessed on: 2022-06-28. [Online]. Available: https://www.bka.de/SharedDocs/Downloads/DE/Publikationen/JahresberichteUndLagebilder/Cybercrime/cybercrimeBundeslagebild2021.html;jsessionid=94BA40DA635B67CE945D1FE45FE8A8E7.live301?nn=28110

[11] Intel Corporation. IPMI Specification, V2.0, Rev. 1.1: Document. Intel. Accessed on: 2022-06-30. [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/servers/ipmi/ipmi-second-gen-interface-spec-v2-rev1-1.html

[12] National Institute of Standards and Technology (NIST). Call for Proposal. Accessed on: 2022-08-23. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf

[13] D. Moody, G. Alagic, D. Apon, D. Cooper, Q. Dang, T. Dang, J. Kelsey, J. Lichtinger, Y.-K. Liu, C. Miller, R. Peralta, R. Perlner, A. Robinson, and D. Smith-Tone, “Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process,” pp. NIST IR 8413–upd1, accessed on: 2023-01-30. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8413-upd1.pdf

[14] SECDER-KONSORTIUM 2022. PROJEKT - SecDER - Störfallinformationssystem für virtuelle Kraftwerke. [Online]. Available: https://secder-project.de/

[15] IMMUNE Konsortium. Immune: Selbstverteidigende netzwerke für resiliente industrie 4.0. Accessed on: 2022-10-31. [Online]. Available: https://immune-projekt.de

[16] A. R. Regenscheid, “Platform Firmware Resiliency Guidelines,” accessed on: 2022-06-28. [Online]. Available: https://www.nist.gov/publications/platform-firmware-resiliency-guidelines

[17] Trusted Computing Group (TCG), “Cyber Resilient Module and Building Block Requirements,” p. 81.

[18] D. Jin, Z. Li, C. Hannon, C. Chen, J. Wang, M. Shahidehpour, and C. W. Lee, “Toward a Cyber Resilient and Secure Microgrid Using Software-Defined Networking,” vol. 8, no. 5, pp. 2494–2504.

[19] M. Medwed, V. Nikov, J. Renes, T. Schneider, and N. Veshchikov, “Cyber Resilience for Self-Monitoring IoT Devices,” in *2021 IEEE International Conference on Cyber Security and Resilience (CSR)*, pp. 160–167.

[20] L. Jäger, D. Lorych, and M. Eckel, “A Resilient Network Node for the Industrial Internet of Things,” in *Proceedings of the 17th International Conference on Availability, Reliability and Security.* ACM, pp. 1–10, accessed on: 2022-09-01. [Online]. Available: https://dl.acm.org/doi/10.1145/3538969.3538989

[21] J. W. Bos, C. Costello, M. Naehrig, and D. Stebila, “Post-Quantum Key Exchange

for the TLS Protocol from the Ring Learning with Errors Problem," in *2015 IEEE Symposium on Security and Privacy*, pp. 553–570.

[22] K. Bürstinghaus-Steinbach, C. Krauß, R. Niederhagen, and M. Schneider, "Post-Quantum TLS on Embedded Systems: Integrating and Evaluating Kyber and SPHINCS+ with mbed TLS," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '20. Association for Computing Machinery, pp. 841–852, accessed on: 2022-05-24. [Online]. Available: https://doi.org/10.1145/3320269.3384725

[23] A. Hülsing, J. Rijneveld, and P. Schwabe, "ARMed SPHINCS Computing a 41 KB Signature in 16 KB of RAM," pp. 446–470.

[24] D. Stebila and M. Mosca, "Post-quantum Key Exchange for the Internet and the Open Quantum Safe Project," in *Selected Areas in Cryptography – SAC 2016*, ser. Lecture Notes in Computer Science, R. Avanzi and H. Heys, Eds. Springer International Publishing, pp. 14–37.

[25] The Open Quantum Safe Project. Home. Open Quantum Safe. Accessed on: 2022-06-29. [Online]. Available: https://openquantumsafe.org/

[26] Open Quantum Safe. Liboqs. Accessed on: 2022-05-12. [Online]. Available: https://github.com/open-quantum-safe/liboqs

[27] I. Duits. The Post-Quantum Signal Protocol : Secure Chat in a Quantum World. Accessed on: 2022-04-21. [Online]. Available: https://essay.utwente.nl/77239/

[28] D. Ott, C. Peikert, and o. workshop participants, "Identifying Research Challenges in Post Quantum Cryptography Migration and Cryptographic Agility," accessed on: 2022-06-27. [Online]. Available: http://arxiv.org/abs/1909.07353

[29] SAFECrypto. About SAFEcrypto – SAFEcrypto. Accessed on: 2022-10-24. [Online]. Available: https://www.safecrypto.eu/welcome/

[30] The Open Quantum Safe Project. Open Quantum Safe: Software for Prototyping Quantum-Resistant Cryptography. Open Quantum Safe. Accessed on: 2022-12-16. [Online]. Available: https://openquantumsafe.org/

[31] Open Quantum Safe. Open Quantum Safe · GitHub. Accessed on: 2022-11-28. [Online]. Available: https://github.com/orgs/open-quantum-safe/repositories?type=all

[32] ——. OQS-OpenSSL_1_1_1. Accessed on: 2022-05-19. [Online]. Available: https://github.com/open-quantum-safe/openssl

[33] ——. OQS-OpenSSH. Accessed on: 2022-12-16. [Online]. Available: https://github.com/open-quantum-safe/openssh

[34] PQCRYPTO. Libpqcrypto: Prerequisites. Accessed on: 2023-01-22. [Online]. Available: https://libpqcrypto.org/install.html

[35] Fraunhofer AISEC. Lazarus: Healing Compromised Devices in the Internet of Small

Things. Accessed on: 2022-05-19. [Online]. Available: https://github.com/Fraunhofer-AISEC/lazarus

[36] R. Spiger. TCG Cyber Resilient Technologies. Accessed on: 2022-05-12. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/TCG-Cyber-Resilient-Technologies-%E2%80%93-Rob-Spiger-Microsoft.pdf

[37] TWI Ltd. Was sind Technology Readiness Levels (TRLs)? Accessed on: 2022-11-03. [Online]. Available: http://live-twi.cloud.contensis.com/locations/deutschland/was-wir-tun/haeufig-gestellte-fragen/was-sind-technology-readiness-levels-trls.aspx

[38] F. Pfeiffer. Induktiv und deduktiv vorgehen in 4 schritten – so geht's! Scribbr. Accessed on: 2023-01-11. [Online]. Available: https://www.scribbr.de/methodik/induktiv-deduktiv/

[39] ——. Fallstudie als methode für deine abschlussarbeit. Scribbr. Accessed on: 2023-01-11. [Online]. Available: https://www.scribbr.de/methodik/fallstudie/

[40] S. Šarić, *Competitive Advantages through Clusters*. Gabler Verlag, accessed on: 2022-10-24. [Online]. Available: http://link.springer.com/10.1007/978-3-8349-3554-0

[41] L. Genau. Die 3 gütekriterien qualitativer forschung erklärt mit beispielen. Scribbr. Accessed on: 2023-01-11. [Online]. Available: https://www.scribbr.de/methodik/guetekriterien-qualitativer-forschung/

[42] ——. Experiment als methode für die abschlussarbeit. Scribbr. Accessed on: 2023-01-11. [Online]. Available: https://www.scribbr.de/methodik/experiment/

[43] J. Eberspächer, Ed., *Sichere Daten, Sichere Kommunikation / Secure Information, Secure Communication: Datenschutz Und Datensicherheit in Telekommunikations- Und Informationssystemen / Privacy and Information Security in Communication and Information Systems*, ser. Telecommunications. Springer Berlin Heidelberg, vol. 18, accessed on: 2022-09-07. [Online]. Available: http://link.springer.com/10.1007/978-3-642-85103-2

[44] E. Barker, "Recommendation for key management:: Part 1 - general," pp. NIST SP 800–57pt1r5, accessed on: 2022-08-30. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf

[45] C. Paulsen and R. Byers, "Glossary of key information security terms," p. NIST IR 7298r3, accessed on: 2022-10-10. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2019/NIST.IR.7298r3.pdf

[46] A. Rein, "Vorlesungsfolien zum kurs 'kryptologie und systemsicherheit'," Wintersemester 2020/2021. [Online]. Available: loesungen_slides_crypto_00_course_intro_bw.pdf

[47] A. Beutelspacher, J. Schwenk, and K.-D. Wolfenstetter, *Moderne Verfahren Der Kryptographie - Von RSA Zu Zero-Knowledge*, 7th ed., ser. STUDIUM. Vieweg+Teubner Verlag | Springer Fachmedien Wiesbaden GmbH 2010, accessed on: 2022-08-08. [Online]. Available: https://link.springer.com/book/9783834812285

[48] A. Beutelspacher, H. B. Neumann, and T. Schwarzpaul, *Kryptografie in Theorie und Praxis: Mathematische Grundlagen für elektronisches Geld, Internetsicherheit und Mobilfunk*, 2005th ed. Vieweg.

[49] K. Quick. Background. Thespian In-Depth Introduction. Accessed on: 2022-10-21. [Online]. Available: https://thespianpy.com/doc/in_depth

[50] ——. Thespian Python Actors. Thespian Python Actors. Accessed on: 2022-06-15. [Online]. Available: https://thespianpy.com/doc/

[51] ——. Actor Overview. Thespian Actors User's Guide. Accessed on: 2023-01-30. [Online]. Available: https://thespianpy.com/doc/using.html#outline-container-hH-f23d674d-ce24-48b4-8ad8-be8eb299e89e

[52] ——. Thespian Features. Thespian In-Depth Introduction. Accessed on: 2022-12-19. [Online]. Available: https://thespianpy.com/doc/in_depth#outline-container-org9bb4305

[53] ——. ActorSystem Implementations: 4 multiprocTCPBase. Thespian Actors User's Guide. Accessed on: 2023-01-30. [Online]. Available: https://thespianpy.com/doc/using.html#outline-container-hH-2a5fa63d-e6eb-43b9-bea8-47223b27544e

[54] C. Paar and J. Pelzl, *Kryptografie Verständlich: Ein Lehrbuch Für Studierende Und Anwender*, ser. eXamen.Press. Springer Vieweg.

[55] D. Wätjen, *Kryptographie: Grundlagen, Algorithmen, Protokolle*, 3rd ed. Springer Vieweg.

[56] B. Buchanan. SHAKE Stirs-up Crypto. ASecuritySite: When Bob Met Alice. Accessed on: 2022-10-10. [Online]. Available: https://medium.com/asecuritysite-when-bob-met-alice/shake-stirs-up-crypto-7d87f3cf39f4

[57] M. J. Dworkin, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions," p. NIST FIPS 202, accessed on: 2022-08-29. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf

[58] L. Yin, B. Fang, Y. Guo, Z. Sun, and Z. Tian, "Hierarchically defining Internet of Things security: From CIA to CACA," vol. 16, no. 1, p. 1550147719899374, accessed on: 2022-08-23. [Online]. Available: https://doi.org/10.1177/1550147719899374

[59] M. Bedner and T. Ackermann, "Schutzziele der IT-Sicherheit," no. 05/2010. [Online]. Available: https://link.springer.com/article/10.1007/s11623-010-0096-1

[60] M. Roetteler, M. Naehrig, K. M. Svore, and K. Lauter, "Quantum resource estimates for computing elliptic curve discrete logarithms," accessed on: 2023-01-27. [Online]. Available: http://arxiv.org/abs/1706.06752

[61] C. Q. Choi. IBM Unveils 433-Qubit Osprey Chip - IEEE Spectrum. Accessed on: 2023-01-27. [Online]. Available: https://spectrum.ieee.org/ibm-quantum-computer-osprey

[62] I. T. L. Computer Security Division. Selected Algorithms 2022 - Post-Quantum Cryptography | CSRC | CSRC. CSRC | NIST. Accessed on: 2022-08-18. [Online].

Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022

[63] S. Ricci, L. Malina, P. Jedlicka, D. Smékal, J. Hajny, P. Cibik, P. Dzurenda, and P. Dobias, "Implementing CRYSTALS-Dilithium Signature Scheme on FPGAs," in *The 16th International Conference on Availability, Reliability and Security*. ACM, pp. 1–11, accessed on: 2022-08-29. [Online]. Available: https://dl.acm.org/doi/10.1145/3465481.3465756

[64] P. Schwabe. Dilithium. Accessed on: 2022-08-18. [Online]. Available: https://pq-crystals.org/dilithium/index.shtml

[65] J. Hoffstein, J. Pipher, and J. H. Silverman, "NTRU: A ring-based public key cryptosystem," in *Algorithmic Number Theory*, ser. Lecture Notes in Computer Science, J. P. Buhler, Ed. Springer, pp. 267–288.

[66] About Falcon. Falcon. Accessed on: 2022-08-18. [Online]. Available: https://falcon-sign.info/

[67] J.-P. Aumasson, D. J. Bernstein, W. Beullens, C. Dobraunig, M. Eichlseder, S. Fluhrer, S.-L. Gazdag, A. Hülsing, P. Kampanakis, S. Kölbl, T. Lange, M. M. Lauridsen, F. Mendel, R. Niederhagen, C. Rechberger, J. Rijneveld, P. Schwabe, and B. Westerbaan, "Sphincs+-r3.1-specification.pdf," accessed on: 2022-08-18. [Online]. Available: https://sphincs.org/data/sphincs+-r3.1-specification.pdf

[68] B. Lapid and A. Wool, "Cache-Attacks on the ARM TrustZone implementations of AES-256 and AES-256-GCM via GPU-based analysis," accessed on: 2022-12-06. [Online]. Available: https://eprint.iacr.org/2018/621

[69] National Institute of Standards and Technology (NIST). Announcement: The End of the 3rd Round - the First PQC Algorithms to be Standardized. Accessed on: 2022-08-30. [Online]. Available: https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/G0DoD7lkGPk?pli=1

[70] S. Bai, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS-Dilithium," p. 38, accessed on: 2022-08-18. [Online]. Available: https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf

[71] P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, "Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU," p. 67, accessed on: 2022-08-18. [Online]. Available: https://falcon-sign.info/falcon.pdf

[72] C. Schönig. Post-quantum-kryptographie: Sichere verschlüsselung trotz quanten-computer. Digitale Welt. Accessed on: 2022-12-15. [Online]. Available: https://digitaleweltmagazin.de/post-quantum-kryptographie-sichere-verschluesselung-trotz-quanten-computer/

[73] M. T. published. Quantum Computers Could Crack Bitcoin Security by the 2030s. Tom's Hardware. Accessed on: 2022-12-15. [Online]. Available:

https://www.tomshardware.com/news/quantum-computer-development-could-put-bitcoin-security-at-risk-by-the-2030s

[74] R. Preston, "Applying Grover's Algorithm to Hash Functions: A Software Perspective," accessed on: 2022-12-15. [Online]. Available: http://arxiv.org/abs/2202.10982

[75] Microsoft Research, "RIoT Reference Architecture," Microsoft, accessed on: 2022-05-12. [Online]. Available: https://github.com/microsoft/RIoT

[76] Trusted Firmware. Mbedtls/library at development · Mbed-TLS/mbedtls. GitHub. Accessed on: 2022-08-18. [Online]. Available: https://github.com/Mbed-TLS/mbedtls

[77] M. Laurin, "Cryptographic library for Python with Mbed TLS back end," accessed on: 2022-08-22. [Online]. Available: https://github.com/Synss/python-mbedtls

[78] K. Quick. ActorSystem API. Thespian Actors User's Guide:. Accessed on: 2023-01-30. [Online]. Available: https://thespianpy.com/doc/using.html#outline-container-hH-7d2b4986-95d3-4460-9d19-ea0dcdb34fca

[79] ——. Thespian Features: 5 Concurrency. Thespian In-Depth Introduction. Accessed on: 2023-01-25. [Online]. Available: https://thespianpy.com/doc/in_depth#outline-container-org9bb4305

[80] Beechat Network Systems Ltd. Dilithium-Py. Accessed on: 2023-01-22. [Online]. Available: https://github.com/BeechatNetworkSystemsLtd/dilithium-Py

[81] T. Prest. Falcon.py. Accessed on: 2022-08-19. [Online]. Available: https://github.com/tprest/falcon.py

[82] J. Rijneveld, P. Schwabe, and R. Gonzalez. PySPX: Python bindings for SPHINCS+. Accessed on: 2023-01-22. [Online]. Available: https://pypi.org/project/PySPX/

[83] J. Rijneveld. PySPX. Accessed on: 2023-01-22. [Online]. Available: https://github.com/sphincs/pyspx

[84] PQClean. PQClean. GitHub. Accessed on: 2022-08-23. [Online]. Available: https://github.com/PQClean/PQClean

[85] mupq. Pqm4. Accessed on: 2023-01-25. [Online]. Available: https://github.com/mupq/pqm4

[86] pq-crystals. Dilithium. GitHub. Accessed on: 2023-01-20. [Online]. Available: https://github.com/pq-crystals/dilithium/tree/9dddb2a0537734e749ec2c8d4f952cb90cd9e67b

[87] A. Prakash, H. Amrouch, M. Shafique, T. Mitra, and J. Henkel, "Improving mobile gaming performance through cooperative CPU-GPU thermal management," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6.