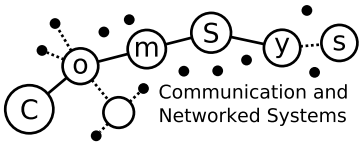
FACULTY OF
COMPUTER SCIENCE



Communication and Networked Systems

Masterarbeit

Interoperable zertifizierende Algorithmen auf eingebetteten Systemen

Moritz Marquardt

Betreuer: Prof. Dr. rer. nat. Mesut Güneş
Betreuender Assistent: MSc. Frank Engelhardt

Institut für Intelligente Kooperierende System, Otto-von-Guericke-Universität Magdeburg

12. April 2022

Zusammenfassung

Zusammenfassung

Der recht neue Ansatz der zertifizierenden Algorithmen zielt darauf ab, die Korrektheit eines algorithmisch ermittelten Ergebnisses mit geringem Rechenaufwand mathematisch nachprüfbar zu machen. Die Herausforderungen sind nun einerseits die Erstellung solcher Algorithmen an sich, sowie andererseits die einheitliche und wiederverwendbare Implementierung dieser, sodass ein solcher Algorithmus an möglichst vielen Stellen zum Einsatz kommen kann. Auf letzteren Punkt bezieht sich diese Arbeit - dabei werden Best Practices ermittelt sowie ein Framework implementiert, mit dem Ziel, die Nutzbarkeit von zertifizierenden Algorithmen auf beliebigen stark eingeschränkten Systemen (vorrangig auf eingebetteten Systemen) auch in komplexen Aufbauten sicherzustellen.

Abstract

The rather new approach of certifying algorithms makes the correctness of one result of an algorithm mathematically verifiable with low computational effort. The challenges are on the one hand the creation of such algorithms themselves, and on the other hand the uniform and reusable implementation of these, so that such an algorithm can be used in as many places as possible. The latter point is the focus of this work - best practices are identified and a framework is implemented, all with the goal of ensuring the usability of certifying algorithms on arbitrary highly constrained systems (primarily on embedded systems) even in complex setups.

Inhaltsverzeichnis

Abbildungsverzeichnis **vii**

Quellcodeverzeichnis **vii**

Tabellenverzeichnis **viii**

Glossar **viii**

1 Einführung **1**
1.1 Motivation . . . 2
1.2 Zielsetzung der Arbeit . . . 2
1.3 Aufbau der Arbeit . . . 3

2 Methodik **5**
2.1 Analyse der Ausgangsinformationen . . . 6
2.2 Ableitung der Best Practices . . . 6
2.3 Software-Framework als Artefakt . . . 7

3 Hintergrund & Verwandte Arbeiten **9**
3.1 Forschungsstand . . . 9
3.2 Existierende Implementierungen . . . 10
3.3 Einsatzbereiche in eingebetteten Systemen . . . 12

4 Framework für interoperable zertifizierende Algorithmen **15**
4.1 Anforderungsermittlung . . . 15
4.2 Erster Evaluationsschritt . . . 16
4.2.1 Designüberlegungen . . . 16
4.2.2 Funktionsweise & Implementierung . . . 18
4.2.3 Evaluation & Anforderungsermittlung . . . 20
4.3 Zweiter Evaluationsschritt . . . 23
4.3.1 Designüberlegungen . . . 23
4.3.2 Funktionsweise & Implementierung . . . 24
4.3.3 Evaluation & Anforderungsermittlung . . . 25

5 Ergebnisse **29**
5.1 Erfüllung der Anforderungen . . . 29
5.2 Best Practices . . . 30

5.3 Nutzung des Frameworks 31
5.3.1 Nutzung von Algorithmen und Checkern 31
5.3.2 Implementierung eines Algorithmus-Skeletts 32
5.3.3 Implementierung von Algorithmen (Executor) 35
5.3.4 Implementierung von Checkern 35
5.3.5 Formale Verifikation der Checker 35
5.4 Eine Registry für konsistente Datentypen 36
5.5 Vergleich von zertifizierenden Algorithmen mit Alternativen 36

6 Fazit **39**
6.1 Zusammenfassung 39
6.2 Future Work 40

Literaturverzeichnis **41**

Anhang **42**
A.1 Vollständiger Quellcode 43
A.2 Doxygen-Dokumentation von libceral 45

Abbildungsverzeichnis

1.1 Zertifizierender Algorithmus für bipartite Graphen 1

2.1 Darstellung des Design Science Research-Projekts als DSR-Grid 5

3.1 Zahl der Veröffentlichungen zu zertifizierenden Algorithmen auf Google Scholar, analysiert nach Sida [1] 9

4.1 Aufbau des CBOR-Schemas für Iteration 1 und 2 im Vergleich 19
4.2 Schematischer Ablauf einer Berechnung mit dem libceral-Framework, mit Vergleich der zwei Iterationen. Die Datenübertragung zwischen anfragendem zum ausführendem System ist nicht Teil des Frameworks; es kann sich dabei auch um dasselbe System handeln - in diesem Fall ist die (De-)Serialisierung nicht notwendig. 20
4.3 Benchmark-Ergebnisse für CPU und Speicher 27
4.4 Benchmark-Ergebnisse für Dateigröße und Code-Statements 28

Quellcodeverzeichnis

5.1 Beispiel für die Nutzung eines zertifizierenden Algorithmus mit *libceral* . . . 31
5.2 myalgorithm.h: Header-Datei für das Skelett eines Beispielalgorithmus . . . 32
5.3 myalgorithm.c: Skelett eines Beispielsalgorithmus 33
5.4 Beispiel für eine Executor-Funktion 35
5.5 Beispiel für eine Checker-Funktion 35

Tabellenverzeichnis

3.1 Gefundene Implementierungen zertifizierender Algorithmen 12

4.1 Benchmark-Ergebnisse für Ausführungszeit (nur Iteration 2 mit Digitsum und ESP8266) 28

Glossar

Algorithmus Generell ein Ablauf zur Berechnung einer bestimmten Ausgabe aus bestimmten Eingabewerten, hier bei *zertifizierenden* Algorithmen vor allem abgegrenzt vom Checker, der die Ausgabe anhand eines ebenfalls vom Algorithmus zurückgegebenen Witness überprüft. 1

Best Practices Oft branchenspezifische übliche Herangehensweisen und Regeln, welche die Erfüllung bestimmter Ziele erleichtern oder möglich machen. v, 1, 6, 15, 29–31, 39, 40

CBOR Concise Binary Object Representation, ein Standard zur binären Serialisierung von Daten. 18, 19, 21–26, 32

Checker Funktion zur Überprüfung der Ausgabe eines zertifizierenden Algorithmus anhand eines mitgelieferten Witness sowie den Eingabewerten. 1–3, 10, 11, 13, 15–19, 22, 24–26, 30, 32, 35–37, 39

formale Verifikation Eine Strategie der Software-Qualitätssicherung, bei der die Korrektheit eines Algorithmus mathematisch bewiesen wird. Theoretisch perfekte Lösung um die Funktion gemäß Spezifikation zu garantieren, aber auch sehr viel Aufwand. 1, 10, 37, 39

GitHub Software-Forge zur Veröffentlichung von und Arbeit an Projekten mit der Versionsverwaltung „Git“, die häufig unter einer freien Lizenz von jedermann nutzbar sind. Siehe `https://github.com`. 6, 11

Input Knowledge Das Anfangswissen, das bei der DSR-Methodik in die Entwicklung des Artefakts mit einfließt. 6, 9, 15

Makro Ein vom C-Präprozessor ersetzter Alias für ein anderes Stück Code, welcher zuvor mit `#define` definiert wurde. 17, 21–23, 25, 34

OID Object Identifier wie in X660 vom ITU-T spezifiziert. 22, 24, 26, 32, 36

Output Knowledge Das wissenschaftliche Ergebnis der DSR-Methodik, welches die wissenschaftliche Fragestellung löst. 5, 39

Preconditions Vorbedingungen, die erfüllt sein müssen, damit ein Algorithmus auf eine Eingabe angewendet werden kann; beispielsweise erlaubte Wertebereiche, Größen von Datentypen, etc. 30

Pull Request Anfrage auf Übernahme einer Änderung in ein Projekt über ein Software-Forge wie GitHub. 36

Shared Library Ein Weg, bereits kompilierte Softwarebibliotheken aus anderen Projekten aufzurufen, und so auf einem System unabhängig vom eigentlichen Projekt verwalten zu können. 22, 25

Software-Framework Strukturierte Sammlung von Funktionen, oft in Form einer Softwarebibliothek, die es über eine API erleichten, eine bestimmte häufig vorkommene Aufgabe in einem Softwareprojekt umzusetzen. 1, 3, 5–7, 10, 15, 29, 39

Solution Das finale Artefakt, welches bei der DSR-Methodik entwickelt wurde, und die Problemstellung in der Industrie löst. 5, 39

Unit-Test Eine Strategie der Software-Qualitätssicherung, bei der Algorithmen mit Beispielwerten aufgerufen werden, deren korrektes Ergebnis bekannt ist, wodurch die wahrscheinlichsten Fehlerfälle abgefangen werden können. 1, 10, 25, 36, 39

Witness Von einem zertifizierenden Algorithmus zurückgegebener zusätzlicher Wert, mit dessen Inhalt der dazugehörige Checker beweisen kann, dass das Ergebnis der Ausführung korrekt ist. 1, 11, 16, 18, 20, 25, 30, 32, 34–36

zertifizierender Algorithmus Nach McConnell et al. [2, Kapitel 1]: „Ein zertifizierender Algorithmus ist ein Algorithmus, der mit jedem Ergebnis zusätzlich ein Zertifikat oder Witness produziert, das beweist, dass dieses spezielle Ergebnis nicht durch einen Bug kompromittiert ist. Durch die Untersuchung dieses Witness mit einem passenden Checker kann der Nutzer sich davon überzeugen, dass das Ergebnis korrekt ist, oder es als fehlerhaft zurückweisen.“. 1

KAPITEL 1

Einführung

Sie sollen gleichermaßen ein Ersatz für Unit-Tests sowie für formale Verifikation von Programmen sein: zertifizierende Algorithmen. Doch ist das realistisch? Was sind die Vor- und Nachteile bei der Nutzung auf eingebetteten Systemen? Und wie lässt sich diese Nutzung so einfach wie möglich machen? All dies soll in dieser Arbeit analysiert werden, das Ergebnis ist ein Satz aus Best Practices zusammen mit einem Software-Framework, welches vor allem auf den meisten eingeschränkten Hardwareplattformen die Nutzung von zertifizierenden Algorithmen einfach und flexibel möglich machen soll.

Beginnen wir mit der wichtigsten Frage: was sind zertifizierende Algorithmen überhaupt? Nach McConnell et al. [2, Kapitel 1] werden sie folgendermaßen definiert:

> Ein zertifizierender Algorithmus ist ein Algorithmus, der mit jedem Ergebnis zusätzlich ein Zertifikat oder Witness produziert, das beweist, dass dieses spezielle Ergebnis nicht durch einen Bug kompromittiert ist. Durch die Untersuchung dieses Witness mit einem passenden Checker kann der Nutzer sich davon überzeugen, dass das Ergebnis korrekt ist, oder es als fehlerhaft zurückweisen.

Grundsätzlich lassen sie sich mit einer Proberechnung vergleichen, wie man sie schon aus der Grundschul-Mathematik kennt, allerdings mit zusätzlichen Informationen aus dem Rechenweg in Form des Witness, wodurch sie gut auch für deutlich komplexere Algorithmen geeignet sind. Der Algorithmus zur Überprüfung („Checker“ genannt) ist dabei zumeist deutlich einfacher und effizienter als der eigentliche Algorithmus, und kann somit einfacher beispielsweise auch formal verifiziert werden.

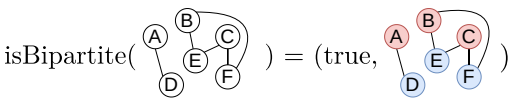


Abbildung 1.1: Zertifizierender Algorithmus für bipartite Graphen

Ein einfaches Beispiel aus [2, Kapitel 2.1] ist, zu überprüfen, ob ein Graph bipartit ist, also in zwei Gruppen aus Knoten einteilbar ist, bei denen jede Kante von einer Gruppe zur anderen führt: wenn jeder Punkt in eine von zwei Farben eingeordnet werden kann, sodass es keine Kante zwischen Punkten der gleichen Farbe gibt, so ist der Graph bipartit.

Das Aufwendige an diesem Algorithmus ist die Unterteilung in die zwei Farben; steht eine solche Unterteilung jedoch als Zertifikat zur Verfügung wie in Abbildung 1.1, ist es leicht zu beweisen, dass die Unterteilung korrekt und der Graph damit bipartit ist (für nicht bipartite Graphen wird in diesem Fall allerdings ein etwas anderer Beweis benötigt).

In der Realität können diese Algorithmen vor allem überall da eingesetzt werden, wo eine Fehlentscheidung kritische Konsequenzen hätte - es muss jedoch immer klar sein, was bei einer solchen zu tun ist: Man kann beispielsweise die Berechnung erneut durchführen (gegen Hardwarefehler und evtl. bösartige Eingriffe), alternative Algorithmen als Fallback nutzen oder einen Mensch eingreifen lassen.

Auch die sichere Auslagerung von Berechnungen auf leistungsfähigere Systeme ist damit möglich; die Integrität der Daten kann so selbst ohne Verschlüsselung gewährleistet werden. Dieses Szenario ist besonders bei eingebetteten Systemen und IoT-Geräten interessant.

Fehler müssen jedoch passieren dürfen - beispielsweise für zeitkritische Anwendungen sind zertifizierende Algorithmen darum weniger geeignet. Im Idealfall können je nach Projekt als Kompromiss auch mehrere Maßnahmen kombiniert werden.

1.1 Motivation

Die Forschung für zertifizierende Algorithmen wird zwar langsam aber stetig mehr (siehe Abschnitt 3.1), jedoch werden die meisten Algorithmen nur theoretisch oder beispielhaft implementiert. Der Einsatz in eigener Software erfordert so einen großen Eigenanteil - zumeist müssen die Algorithmen mit den genutzten Programmiersprachen und Werkzeugen vollständig neu implementiert werden, wobei einige Dinge zu beachten sind, gerade wenn der Algorithmus auf einem anderen System ausgeführt werden soll als der Checker.

Dass sich hier grundsätzliche Fehler einschleichen können, ist offensichtlich - wenn es viele Implementierungen eines Algorithmus gibt, ist es schwierig, gemeinsam Fehler zu finden und zu beheben; gleichzeitig ist es aber auch nicht so einfach möglich, eine einzige Implementierung so zu entwickeln, dass sie für jegliche Anwendungsfälle perfekt geeignet ist.

Damit zertifizierende Algorithmen möglichst fehlerfrei und vielseitig eingesetzt werden können, bietet sich ein Framework an, mit dem möglichst viele Einsatzszenarien abgedeckt sind. Zudem ist es allerdings auch wichtig, zu verstehen, welche Anforderungen wie erfüllt werden können, damit auch sichere und zuverlässige eigene Implementierungen möglich sind.

1.2 Zielsetzung der Arbeit

Mit dieser Arbeit soll eine Basis für die Nutzung zertifizierender Algorithmen geschaffen werden, die den regulären Einsatz selbiger in Softwareprojekten aller Art, vor allem aber auf eingebetteten Systemen, ermöglicht.

Um dies umzusetzen, sollen folgende Forschungsfragen beantwortet werden, wobei der Schwerpunkt auf Punkt FF2 liegt:

FF1 Für welche Aufgaben in den Bereichen IoT & Eingebettete Systeme können zertifizierende Algorithmen hilfreich sein? Wie unterscheiden sich alternative Ansätzen?

FF2 Welche Aspekte müssen generell bei der Implementierung von zertifizierenden Algorithmen beachtet werden, vor allem im Kontext von eingebetteten Systemen?

FF3 Wie aufwendig ist der Einsatz eines Software-Frameworks in verschiedenen Projekten, um damit a) einen zertifizierenden Algorithmus zu implementieren, und b) einen zertifizierenden Algorithmus über eine Netzwerkschnittstelle auf einem anderen System auszuführen? Welcher Overhead entsteht dabei?

Um die Antworten auf diese Fragen zu finden, sowie die spätere Nutzung so einfach wie möglich zu machen, soll zum einen bestehende Literatur analysiert werden, und daraufhin ein Software-Framework entwickelt und evaluiert werden. Das wichtigste Ziel der Frameworkentwicklung soll dabei sein, auf stark eingeschränkten Geräten möglichst flexibel auch komplexe zertifizierende Algorithmen implementieren und nutzen zu können.

Die so implementierten Algorithmen sollen dabei auch interoperabel in anderen Projekten (beispielsweise auf anderer Hardware oder mit anderen Toolchains) genutzt werden können. Zudem soll sich der Algorithmus auf einem anderen Gerät als der Checker ausführen lassen.

Das Minimum Viable Product ist demnach ein Framework, mit dem zumindest ein einfacher zertifizierender Algorithmus implementiert werden kann, bei dem der Algorithmus auf einem anderen Gerät als der Checker läuft, und besonders der Checker auf einem stark eingeschränkten Gerät (z. B. ESP8266) funktioniert.

Als optionale weitere Ziele kommt ein einfaches Einsatzbeispiel in einem Node-Cluster infrage (also eine Integration mit bestehenden Ansätzen zur Kommunikation, Lastverteilung, usw.), sowie die Unterstützung einfacher verteilter zertifizierender Algorithmen nach [3].

Explizit kein Ziel ist die Vorgabe von Sprachen und Werkzeugen bei der Entwicklung der Algorithmen selbst, da dies unabhängig vom Framework bleiben soll, um auch möglichst viele bestehende Algorithmen mit dem Framework kompatibel machen zu können.

1.3 Aufbau der Arbeit

Nach der Einführung soll in Kapitel 2 die Methodik beschrieben werden, durch welche unter anderem mit Design Science Research das Framework entwickelt wird, aus dessen Ergebnissen schließlich die Antworten auf die Forschungsfragen abgeleitet werden.

In Kapitel 3 werden daraufhin verwandte Arbeiten untersucht, um eine Grundlage für die Entwicklung der Forschung in dieser Arbeit zu schaffen.

Diese Grundlagen werden schließlich in Kapitel 4 genutzt, um zuerst Anforderungen festzulegen und daraus ein Framework zu entwickeln, welches genutzt werden kann, um zertifizierende Algorithmen zu implementieren und zu evaluieren. Dies erfolgt in drei Iterationen, in denen jeweils Anforderungen umgesetzt werden, und im Anschluss durch eine Evaluation die Qualität und mögliche zusätzliche Anforderungen festgestellt werden.

Die Ergebnisse werden schließlich in Kapitel 5 zusammengefasst und in der Form von Best Practices zu einer Empfehlung aufbereitet, die für die interoperable Entwicklung von zertifizierenden Algorithmen genutzt werden kann.

Neben dem abschließenden Fazit in Kapitel 6 werden dort mögliche Fragen und Aufgaben für zukünftige Forschung und die Verbesserung des Frameworks aufgeführt.

KAPITEL 2

Methodik

Im Folgenden soll die Methodik, mit der die Forschungsfragen beantwortet werden sollen, näher erläutert werden. Da die Forschung durch die Entwicklung und Evaluation eines Software-Frameworks getrieben werden soll, fällt die Wahl des grundlegenden Methodikansatzes hierbei auf Design Science Research [4] in Anlehnung an das methodische Framework von Peffers et al. [5]: das Software-Framework wird als Design-Artefakt entwickelt und evaluiert, um Wissen über die Umsetzung ähnlicher Projekte zu gewinnen.

In Abbildung 2.1 sind hierfür die Rahmenbedingungen als DSR-Grid [6] dargestellt. An dem so eingegrenzten Prozess bedient sich die gesamte Arbeit, um die Solution als technisches sowie die Output Knowledge als wissenschaftliches Ergebnis zu erzielen.

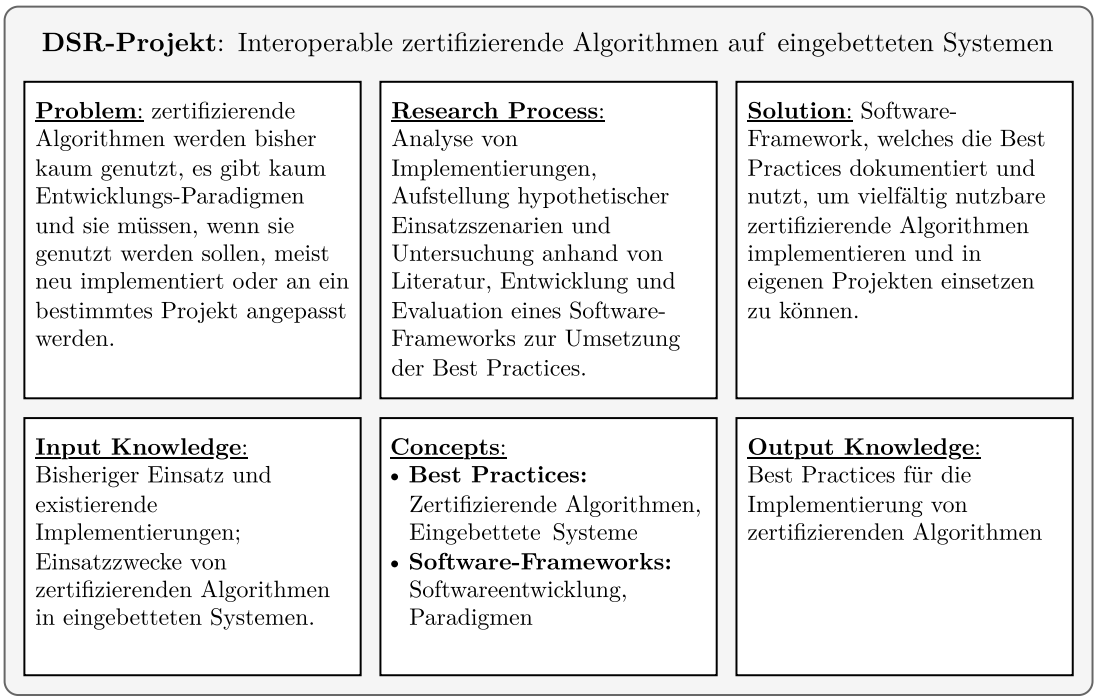


Abbildung 2.1: Darstellung des Design Science Research-Projekts als DSR-Grid

2.1 Analyse der Ausgangsinformationen

Zunächst soll dafür in Kapitel 3 die Input Knowledge betrachtet und aufbereitet werden. Zentrale Fragestellung dabei ist, wie zertifizierende Algorithmen bisher genutzt werden, und ob diese so auch auf eingebetteten Systemen angewendet werden können, beziehungsweise welche Herausforderungen dabei entstehen.

Als Ausgangsinformationen werden vorrangig wissenschaftliche Texte genutzt, wobei als Einstiegspunkt dafür vor allem [2] als eine der wegweisenden Publikationen im Bereich der zertifizierenden Algorithmen dient.

Die Suche nach weiteren Texten beruht vorrangig auf dem Suchbegriff „certifying algorithm(s)“ (als zusammenhängender Begriff) in den wissenschaftlichen Suchmaschinen IEEE Xplore, ACM Digital Library und Google Scholar. Aufgrund der geringen Ergebniszahl wurden die Ergebnisse nicht strategisch näher eingegrenzt, sondern manuell nach ihrer Relevanz eingeordnet - der Fokus liegt dabei vor allem auf der Einführung neuer (Software-)Konzepte, sodass eine Vielzahl von Ergebnissen, die lediglich bestehende Algorithmen zertifizierend umsetzen, nicht mit einbezogen werden.

Zusätzlich werden bestehende Implementierungen untersucht, darunter unter anderem die Bibliothek LEDA [7], sowie alle fünf (!) zum Zeitpunkt der Arbeit auffindbaren GitHub-Repositories zu zertifizierenden Algorithmen.

Da es kaum existierende Forschung zu zertifizierenden Algorithmen mit Fokus auf eingebettete Systeme gibt, ist es nicht sinnvoll möglich, existierende Einsatzszenarien zu untersuchen. Stattdessen werden mögliche Szenarien als Hypothese formuliert, für welche dann anhand der weiteren Ausgangsinformationen geprüft wird, ob die Nutzung zertifizierender Algorithmen für den jeweiligen Fall eine sinnvolle Alternative zu bisherigen Test- und Verifikationsmethoden sein können.

2.2 Ableitung der Best Practices

Aus allen so gesammelten Ausgangsinformationen sowie den Einsatzszenarien werden daraufhin einige Best Practices abgeleitet: daran wie bisher gearbeitet wird, welche Probleme dabei ermittelt werden können und welche Gemeinsamkeiten sich feststellen lassen, soll abgeleitet werden, welche Arbeitsweisen zu möglichst interoperablen und mit vielen Projekten kompatiblen Implementierungen führen können.

Best Practices werden dabei definiert als in einer Branche übliche Herangehensweisen und Regeln, welche die Erfüllung bestimmter Ziele grundsätzlich in vielfältigen Projekten erleichtern können.

Später werden auch aus der Evaluation des entwickelten Software-Frameworks neue empfohlene Kandidaten für Best Practices abgeleitet; dies erfolgt auf der Grundlage des Framework-Designs bei der Evaluation: wenn eine Design-Entscheidung innerhalb einer Iteration einen positiven Einfluss hatte, und grundsätzlich auch außerhalb des Frameworks angewendet werden kann, handelt es sich dabei um einen Kandidaten für eine Best Practice. Diese Kandidaten sind Empfehlungen, die nachweislich genutzt werden können, um ein Ziel leichter erfüllen zu können, werden schließlich jedoch nur bei einer langfristigen Etablierung in anderen Projekten zu tatsächlichen Best Practices.

2.3 Software-Framework als Artefakt

Die Hauptkomponente des Research-Prozesses ist schließlich die iterative Entwicklung eines Software-Frameworks, mit dem diese Best Practices umgesetzt werden können, und zertifizierende Algorithmen von jedem implementierbar beziehungsweise nutzbar gemacht werden können. Mit jeder Iteration werden dabei - sofern relevant - neue Best Practices erschlossen und das Framework verbessert, sodass am Ende sowohl die wichtigsten Herangehensweisen bekannt sind, als auch das Software-Framework zur Nutzung zur Verfügung steht.

Die Implementierung erfolgt dabei mit Git und C derart, dass sie in den meisten Projekten genutzt werden kann, und wird dafür unter einer freien Softwarelizenz zur Verfügung gestellt. Die Möglichkeiten zur Integration verschiedener anderer Programmiersprachen wird dabei stichprobenweise geprüft, und das Framework durch ausreichend flexible vorspezifizierte Schnittstellen so gestaltet, dass eine Reimplementierung in einer anderen Programmiersprache grundsätzlich einfach möglich sein sollte.

Für die nach jedem Implementierungsschritt erfolgende Evaluation des Software-Frameworks werden vorher Ziele spezifiziert, welche sich an der möglichst flexiblen und effizienten Nutzung orientieren (Hardwareanforderungen, Interoperabilität, etc.). Nach jedem Implementierungsschritt erfolgt eine Evaluation, in der die Erreichung der Ziele festgestellt wird. Für nicht erreichte Ziele wird soweit möglich eine Herangehensweise für die nächste Iteration festgelegt; zusätzlich können nach jeder Iteration neue Ziele festgelegt werden, die aufgrund einer während der Implementierung festgestellten Einschränkung sinnvoll sind. So wird mit das Framework strategisch mit jeder Iteration verbessert, bis es am Ende die zu Beginn gestellten Anforderungen erfüllt.

KAPITEL 3

Hintergrund & Verwandte Arbeiten

Im Folgenden werden die Hintergründe von zertifizierenden Algorithmen betrachtet, verwandte Arbeiten und existierende Implementierungen analysiert und schließlich die Nutzbarkeit solcher Algorithmen auf eingebetteten Systemen in verschiedenen Einsatzszenarien evaluiert. Somit stellt dieses Kapitel die Input Knowledge aus der DSR-Grid dar.

3.1 Forschungsstand

Zertifizierende Algorithmen stehen noch ziemlich am Anfang der Forschung - eine Analyse mit [1] offenbart, wie in Abbildung 3.1 zu sehen, dass der Begriff „certifying algorithm(s)“ in den letzten Jahren zwar immer mehr in Veröffentlichungen auftaucht, jedoch mit unter 60 Treffern pro Jahr immer noch sehr spärlich vertreten ist - zur Einordnung: „embedded systems“ kommen allein im Jahr 2021 auf über 20.000 Treffer, „formal verification“ immerhin auf über 6.000. Zu zertifizierenden Algorithmen in Kombination mit eingebetteten Systemen gibt es noch einmal deutlich weniger Veröffentlichungen.

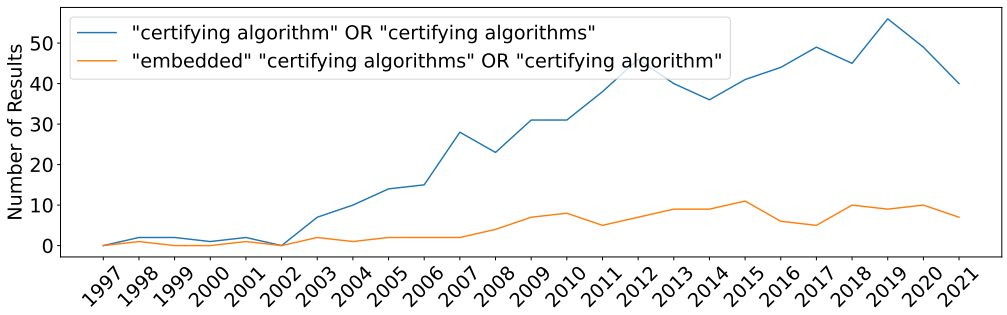


Abbildung 3.1: Zahl der Veröffentlichungen zu zertifizierenden Algorithmen auf Google Scholar, analysiert nach Sida [1]

Eine ausgezeichnete Grundlage schaffen dabei McConnell et al. [2] mit dem Ergebnis, dass zertifizierende Algorithmen grundsätzlich eine universelle Möglichkeit zur Qualitätssicherung von Software sind. Laut der dort durchgeführten Survey existieren bereits über 100

verschiedene zertifizierende Algorithmen, als Implementierung wird jedoch vor allem die Bibliothek LEDA eingegangen, die in Abschnitt 3.2 auch näher betrachtet wird. Damit wird klar, dass zertifizierende Algorithmen durchaus das Potenzial haben, von einer Nischentechnologie zu einer echten Alternative zu Unit-Tests und formaler Verifikation zu werden - die Verbreitung geht jedoch offenbar dennoch recht schleppend voran.

Darauf folgt der Versuch von Alkassar et al. [8], einerseits die (meist einfacheren) Checker formal zu verifizieren, und dies auf einige Algorithmen der recht weit verbreiteten Algorithmen-Bibliothek LEDA anzuwenden. Ergebnis ist, dass dies mit dem Werkzeug VCC recht einfach anhand des C-Codes sowie des im Paper vorgestellten Modells möglich ist, mittlerweile ist die Webseite `http://vcc.codeplex.com/` jedoch nicht mehr erreichbar - der Prozess sollte mit ähnlichen Werkzeugen jedoch ähnlich funktionieren.

Hübschle-Schneider und Sanders [9] betrachten daraufhin parallelisierbare Algorithmen, die entweder auf mehreren Kernen eines Systems oder auch über verschiedene Systeme verteilt ausgeführt werden. Dabei gibt es nicht direkt einen Fokus auf zertifizierenden Algorithmen: es ist vor allem relevant, dass das Ergebnis korrekt ist, nicht dass dies auch im Nachhinein bewiesen werden kann - damit kommt neben den Checkern zertifizierender Algorithmen auch „Algorithm-Based Fault Tolerance“ als Option infrage, ein Ansatz, welcher mit Prüfsummen arbeitet. Implementiert wurde dies im Framework Thrill[1], welches zur Verarbeitung von großen Datenmengen dient. Dabei konnte mit den evaluierten Modellen wohl bei unter 5 % Overhead ein korrektes Ergebnis garantiert werden.

Im Jahr 2020 hat Völlinger [3] schließlich *verteilte* (statt ansonsten *sequentielle*) zertifizierende Algorithmen für den Einsatz in verteilten Systemen evaluiert, um „Lösungen für die grundlegenden Probleme, die durch eine Verteilung der Berechnung entstehen“ [3, 4.2.1] zu liefern. Interessant ist hierbei auch die Unterscheidung zwischen terminierenden und nichtterminierenden Algorithmen: bei ersteren lassen sich zertifizierende Algorithmen sehr viel einfacher anwenden, da bei einem Algorithmus, der nicht terminiert, nicht klar ist, wann bzw. welches Ergebnis überprüft werden soll - bei verteilten Algorithmen stellt die Feststellung der Terminierung dabei einen nicht zu vernachlässigenden Aufwand dar [3, 5.3]. Für die Evaluation wurde ein Software-Framework auf Grundlage des Beweisassistenten „Coq“ entwickelt - das besondere dabei ist, dass sowohl Algorithmus als auch Checker über ein Netzwerk verteilt sein können - jeder Knoten eines vorgeblich bipartiten Graphs kann also beispielsweise selbst bei allen Nachbarknoten feststellen, ob diese eine andere Farbe haben als er selbst. Dies erfordert allerdings deutlich mehr Vertrauen in alle Bestandteile des Netzwerks im Vergleich zu sequentiellen Checkern.

3.2 Existierende Implementierungen

Um die Anforderungen an ein Framework ermitteln zu können, ist es zunächst notwendig, zu verstehen, wie in bisherigen zertifizierenden Algorithmen gearbeitet wird. Dafür sollen im Folgenden einige Implementierungen zertifizierender Algorithmen verschiedener Autoren betrachtet werden - im Idealfall mit einem Fokus auf eingebettete oder anderweitig eingeschränkte Systeme, jedoch gibt es hier, ähnlich wie in der Forschung, noch kaum Projekte.

Die wohl bekannteste Softwarebibliothek, die zwar noch nicht den Namen „zertifizierende

[1] `https://project-thrill.org/`

Algorithmen“ nutzt, jedoch zu vielen Algorithmen einen Checker mitliefert, ist LEDA [7]. Diese steht allerdings unter einer proprietären Lizenz, und ist aktuell kaum noch im Internet aufzufinden - die erste Version stammt von 1990; seit 2022 ist die Webseite nicht mehr erreichbar, allerdings wurde 2020 noch Version 6.6 veröffentlicht. Sie scheint jedoch auch über ihr spezielles Gebiet (Graphen & Geometrie) hinaus für die meisten Algorithmen eine gute Grundlage zu bieten. Die Quelldateien sind zwar nicht frei verfügbar, es kann jedoch die Dokumentation [10] genutzt werden, um die Ansätze und Paradigmen zu untersuchen.

Ein Beispiel, welches bei LEDA betrachtet werden kann, ist der kürzeste Pfad in einem Graphen[2]. Hierbei ist auffällig, dass der Checker `CHECK_SP_T` für alle Shortest-Path-Algorithmen genutzt wird, und dabei kein Witness benötigt, sondern lediglich Ein- und Ausgabewerte überprüft. Andere Algorithmen in LEDA wie der Algorithmus zur „Maximum Cardinality Matchings in General Graphs“[3] liefern ein echtes Witness (hier `OSC`) mit zurück, und lassen sich somit eindeutig als zertifizierende Algorithmen klassifizieren.

Auffällig ist dabei zudem, dass es nicht immer einen konsistenten Aufbau von Algorithmus oder Checker gibt: der Checker von Maximum Cardinality Matchings gibt beispielsweise das Ergebnis des Checks als `bool` zurück, während der Checker der Shortest-Path-Algorithmen laut Dokumentation das komplette Programm abbricht wenn ein Fehler gefunden wurde [10, Abschnitt „Shortest Path Algorithms“].

Anhand von Beispielen, die auch in der LEDA-Bibliothek zur Verfügung stehen, entwickeln Alkassar et al. [8] daraufhin formal verifizierte Checker. Dabei werden teilweise auch Lücken in der Implementierung der Checker von LEDA aufgedeckt, und generell eine gute Grundlage für die Verifizierung von Checkern geschaffen.

Für Algorithmen für kürzeste Pfade in Graphen wird hier jedoch ebenfalls kein zusätzliches Witness verwendet - streng genommen handelt es sich dabei also nicht um einen zertifizierenden Algorithmus, wobei die Definition hier nicht allzu genau genommen werden sollte: derselbe Algorithmus wäre danach beispielsweise zertifizierend, wenn er statt dem Pfad selbst nur die Länge zurückgibt, und den Pfad als Witness nutzt.

Einen guten Überblick über die Nutzung von Softwarephänomenen aller Art bietet natürlich auch GitHub[4] - hier gibt es jedoch leider zum Suchbegriff „certifying algorithms“ nur exakt vier Algorithmen, plus noch ein Ergebnis zu „certifying distributed algorithm“. An die meisten Projekte ist zudem ein wissenschaftliches Paper geknüpft, eine tatsächliche Nutzung in der Industrie lässt sich daraus nicht ableiten. In Tabelle 3.1 sind die gefundenen Implementierungen aufgelistet, zusammen mit der genutzten Programmiersprache und dem Jahr der letzten Aktualisierung.

Wie man sehen kann, variiert die Wahl der Programmiersprachen dabei enorm: C++, Python, Java, Coq und Agda sind dabei - fünf verschiedene Programmiersprachen bei fünf Ergebnissen. Vor allem die vielen unterschiedlichen Sprachen, aber auch die Tatsache, dass alle Projekte bis auf das von Völlinger nicht frei lizenziert sind, offenbaren ein ziemliches Problem: keiner dieser Algorithmen lässt sich mit wirklich einem anderen gemeinsam nutzen, und die Algorithmen lassen sich effektiv kaum in einem bestehenden oder geplanten Projekt einsetzen.

[2] `http://www.algorithmic-solutions.info/leda_manual/shortest_path.html`, abgerufen 1. März 2022
[3] `http://www.algorithmic-solutions.info/leda_manual/mc_matching.html`, abgerufen 1. März 2022
[4] `https://github.com`, abgerufen 1. März 2022

Projekt/Repository	**Algorithmus**	**Sprache**	**Update**
adrianN/edge-connectivity	3-Edge-Connectivity	Python 2	2012
adrianN/Triconnectivity [11]	3-Edge-Connectivity	C++/LEDA	2011
rodrigogribeiro/regexpequiv	Äquivalenz von regulären Ausdrücken	Agda	2015
SleekPanther/3-sat-certifier	3-SAT-Problem	Java	2017
voellinger/verified-certifying-distributed-algorithms [3]	Is-Bipartite	Coq	2019
	Shortest Path	Coq	2019
LEDA [7]	viele Algorithmen für Graphen, Geometrie und mehr	C	2020

Tabelle 3.1: Gefundene Implementierungen zertifizierender Algorithmen

Die Implementierungen von Völlinger und LEDA versuchen dabei als einzige Projekte, eine Art Framework bereitzustellen, haben dabei allerdings nicht direkt zertifizierende Algorithmen im Allgemeinen als vorgesehenen Einsatzzweck.

So fokussiert sich Völlinger auf *verteilte* zertifizierende Algorithmen, welche teilweise andere Anforderungen als klassische sequentielle zertifizierende Algorithmen haben. Die Einbindung des recht komplexen Ökosystems der Programmiersprache des Beweisassistenten Coq könnte dabei für viele Projekte eine zusätzliche Herausforderung darstellen - die Transpilierung in unterschiedliche Sprachen ist jedoch möglich[5], die OCaml-Version sollte sich beispielsweise über OMicroB auch auf eingebetteten Systemen ausführen lassen[6].

LEDA implementiert stattdessen direkt die Algorithmen selbst, ohne ein Framework dafür bereitzustellen. Der Framework-Teil beschränkt sich hier vor allem auf Konventionen und Hilfsfunktionen für Datentypen, die bei der Implementierung von Algorithmen allgemein notwendig sind. Dies kann tatsächlich sehr hilfreich sein, ist jedoch dadurch eingeschränkt, dass LEDA nicht unter einer offenen Lizenz verfügbar ist, und bei der kostenlosen Version der Code nicht einsehbar ist. Dazu kommt, dass die Bibliothek auf der offiziellen Webseite nicht mehr beworben wird.

3.3 Einsatzbereiche in eingebetteten Systemen

Die meisten Grundlagen existieren also bereits, jedoch ist die meiste Forschung eher theoretischer Natur - der tatsächliche Einsatz in der Industrie ist noch sehr selten, und nur wenige Forschungsergebnisse stehen in einer in vielfältigen Projekten einsetzbarer Form zur Verfügung. Das Ziel dieser Arbeit - ein Framework für die plattformunabhängige Nutzung von zertifizierenden Algorithmen zu schaffen - soll es darum vor allem möglich machen, zukünftige Forschungsergebnisse auch in der Industrie einsetzen zu können.

Hier stellt sich natürlich die Frage, welche Algorithmen das überhaupt sind, und wann sich generell zertifizierende Algorithmen auf eingebetteter Hardware lohnen - im Folgenden gibt es eine Aufstellung über verschiedene hypothetische Szenarien, zusammen mit den damit verbundenen Einsatzmöglichkeiten und Hindernissen für zertifizierende Algorithmen:

[5] `https://coq.inria.fr/distrib/current/refman/addendum/extraction.html`, abgerufen 11. April 2022
[6] `https://github.com/benmandrew/omicrob-riot-nrf52`, abgerufen 11. April 2022

- **Sensornetzwerke: Auslagerung von aufwendigen Rechenaufgaben**

  Wenn in einem Netzwerk aus verschiedenen Systemen komplexe Algorithmen zum Einsatz kommen, kann es Sinn ergeben, die Ausführung von vielen schwächeren Systemen (die beispielsweise Sensordaten als Eingabewerte nutzen und die Ergebnisse nutzen) auf wenige stärkere Systeme auszulagern.

  Hier könnten zertifizierende Algorithmen zum Einsatz kommen, um die Integrität und Korrektheit der Ergebnisse zu überprüfen - sowohl bei Programm- und Datenübertragungsfehlern, als auch im Fall von böswilligen Angriffen. In Fällen, in denen nur der Integritäts-Aspekt einer verschlüsselten Verbindung relevant ist, könnte ein zertifizierender Algorithmus theoretisch sogar die rechenintensive Verschlüsselung ersetzen, und damit günstigere oder kleinere Hardware ermöglichen.

  Relevant dabei ist, dass die Ergebnisse vom selben System genutzt werden müssen, welches die Berechnung auch angestoßen hat - ansonsten muss auch die Integrität der Eingabedaten validiert werden, womit zertifizierende Algorithmen allein keine Lösung darstellen können.

  Damit beschränkt sich der Einsatzzweck vor allem auf die Aufbereitung von Daten, die primär für das System kritisch sind, von dem sie erhoben wurden - ein Beispiel wäre ein sicherheitskritisches Sensorsystem im Industrial Internet of Things, das Sensoren an einer Maschine nutzt, und diese Daten mit einem komplexen Algorithmus auf einem separaten System auf Fehlerzustände hin auswerten lässt; die Daten sind zur Früherkennung von Fehlern zwar sekundär auch an einer zentralen Stelle relevant, primär ist die Auslösung eines Not-Aus durch das System selbst jedoch die kritischste Komponente. Aus einem oder mehreren vom Checker als falsch eingestuften Ergebnissen könnte hier ebenfalls die Notwendigkeit einer Notabschaltung abgeleitet werden.

  Solche Einsatzszenarien setzen selbstverständlich die Existenz geeigneter zertifizierender Algorithmen voraus, wobei besonders bei moderneren Ansätzen wie Machine Learning noch viel Forschung erforderlich ist.

- **Sensornetzwerke: verteilte Berechnungen**

  Sollen Aufgaben über mehrere Systeme verteilt werden, sind verteilte zertifizierende Algorithmen eine Möglichkeit, welche von Völlinger [3] im Detail betrachtet wird.

  So können beispielsweise bestimmte Eigenschaften eines Netzwerks analysiert werden, indem darauf verteilte Graph-Algorithmen angewendet werden. Auch für manche Probleme im „Fog Computing“ (also viele IoT-Geräte, die vorrangig selbstständig und dezentral zusammenarbeiten und beispielsweise nur zeitweise an ein größeres Netzwerk oder das Internet angeschlossen werden [12]) kann dies eine Lösung darstellen. Hier wären auch zertifizierende Konsensusalgorithmen denkbar, oder beispielsweise die Berechnung von Durchschnittswerten von Sensoren.

  Wichtig ist dabei, dass es deutlich mehr Vertrauen in alle Komponenten geben muss als bei nicht verteilten zertifizierenden Algorithmen, insbesondere wenn auch der Checker verteilt ausgeführt wird.

- **Luft- und Raumfahrt: Validierung redundanter Systeme**

  Zur Validierung von Sensorwerten und Berechnungen kommen hier häufig komplexe redundante Systeme zum Einsatz, bei denen es wichtig ist, Fehler zu erkennen [13].

Wenn ein Fehler seine Ursache in der Datenverarbeitung statt direkt in den von Sensoren aufgezeichneten Daten hat, könnte ein zertifizierender Algorithmus helfen, die korrekte Seite des redundanten Systems abzuschalten oder eine manuelle Steuerung zu erzwingen, statt fehlerhafte Werte zu nutzen.

Ein Beispiel für einen solchen Fehler wären Integer Overflows, welche schon mehrfach zu Problemen geführt haben. Sie waren beispielsweise die Ursache für einen Zwischenfall mit der Weltraumsonde Ariane 5, bei welchem diese sich selbst zerstörte [14], oder für Probleme mit der Boeing 787, bei denen die Motoren während dem Flug einfach ausgehen konnten [15].

Während die Aktion bei der Feststellung des Fehlers von Stelle und möglichen Auswirkungen des Fehlers abhängt (in der Avionik wäre das z. B. oft die Abgabe der Kontrolle an die Piloten), ist die Feststellung des Fehlers essenziell, andernfalls besteht das Risiko, dass beispielsweise bei zwei gleichzeitigen Ausfällen auf das falsche System gewechselt wird. Hierfür wird auch jetzt schon unter anderem Software-basierte Fehlererkennung eingesetzt. [13, 28.4]

Jazequel und Meyer [15] sehen die Unfallursachen in mangelnder Spezifikation, dies ist natürlich auch bei zertifizierenden Algorithmen möglich - diese stellen darum bei solch komplexen Sensorsystemen generell keine alleinige Lösung dar.

- **Autonome Unterwassersysteme bzw. auch Robotik im Allgemeinen: Reaktion unbemannter Systeme auf Sensoreingaben**

  Besonders [16] beschäftigt sich mit diesem Einsatzzweck - ähnlich wie in der Raumfahrt sind diese Systeme oft unbemannt und nicht extern steuerbar. Beispielsweise für die Hindernisvermeidung und Auswertung von Sensordaten kommen hier sehr häufig Algorithmen zum Einsatz, deren fehlende Korrektheit fatale Folgen haben könnte.

  Auch hier wird von Lüth et al. hervorgehoben, dass die reine Verifikation (oder Zertifizierung) nicht ausreicht - es wird eine Kombination von „Verification & Validation“ benötigt, bei der auch die Spezifikation validiert wird. Ebenfalls wird ein System mit mehreren Ebenen vorgeschlagen - die hardwarenächste Ebene muss bestimmte Bedingungen nachweislich erfüllen (z. B. ein Roboter kann nicht in ein Hindernis gesteuert werden), wodurch die höheren Ebenen mit weniger Verifikation auskommen.

- **Analoge Sensoren im Allgemeinen: Korrektheit der Werte**

  Im Idealfall gibt es auch auf der Hardware-Ebene einen Weg, um die Daten eines Sensors verifizieren zu können. Zertifizierende Algorithmen sind hier natürlich keine Lösung, wenn man analoge Daten betrachtet - stattdessen gibt es hier unter dem Namen „selbstüberwachende Sensoren“ eine Lösung mit einem ähnlichen Ziel [17].

  Während die Korrektheit analoger Hardware sich im Gegensatz zu Algorithmen nicht mathematisch beweisen lässt, sind diese Sensoren darauf ausgelegt, ihre Abweichung von der Spezifikation zu erkennen, um so die Korrektheit zu bestätigen.

  Nur wenn alle Sensordaten, die von einem System mit zertifizierenden Algorithmen verarbeitet werden, von selbstüberwachenden Sensoren kommen, kann ohne Einschränkungen davon ausgegangen werden, dass auf ein valides Endergebnis (zumindest im Rahmen der Spezifikation der Sensoren und der Algorithmen) auch wirklich vertraut werden kann.

KAPITEL 4

Framework für interoperable zertifizierende Algorithmen

In diesem Kapitel wird ein Software-Framework spezifiziert, entwickelt und evauliert, mit dem zertifizierte Algorithmen möglichst interoperabel auf Geräten aller Art genutzt werden können. Hauptziel ist dabei (siehe Abschnitt 1.2) sowohl die Unterstützung stark eingeschränkter Geräte, als auch die Implementierung komplexer Algorithmen auf möglichst flexible Art und Weise zu ermöglichen.

Der gesamte so entstandene Code mit Beispielen ist in Abschnitt A.1 zu finden.

Das finale Framework wird schließlich auf `https://codeberg.org/ovgu/libceral` unter dem Namen *libceral* auch unabhängig von dieser Arbeit öffentlich unter der freien Unlicense[1] verfügbar sein.

4.1 Anforderungsermittlung

An die Implementierung von zertifizierenden Algorithmen gibt es einige wichtige Anforderungen, die zur Erreichung der Ziele aus Abschnitt 1.2 notwendig sind. Im folgenden werden diese Anforderungen anhand von Aufgabenstellung und Input Knowledge festgelegt, sodass sie für die Evaluation des Frameworks und schließlich auch die resultierenden Best Practices genutzt werden können. Bei der Evaluation können dabei jeweils auch weitere Probleme und neue Umstände zur Einführung zusätzlicher Anforderungen führen.

A1 Eine wichtige Anforderung ist die **Unabhängigkeit von der gewählten Programmiersprache** - das Framework soll darum mit mehreren Programmiersprachen genutzt werden können, und gegebenenfalls auch in mehreren Programmiersprachen implementiert werden können. Die Definition of Done ist dabei, dass eine Implementierung mit dem gleichen Algorithmus in mindestens drei unterschiedlichen Programmiersprachen funktioniert.

A2 Damit es bei der Auslagerung komplexer Algorithmen auf leistungsstärkere Systeme keinen unnötigen Overhead gibt, **müssen Algorithmus und Checker voneinan-**

[1] `https://unlicense.org/`

der trennbar sein - dies ist vor allem relevant bei Algorithmen mit vielen Abhängigkeiten oder Abhängigkeiten mit bestimmten Systemanforderungen. Projekte müssen also über eine Option einfach ohne die Implementierung des Algorithmus selbst kompiliert werden können.

A3 Gerade für den Einsatz auf eingebetteter Hardware ist darüber hinaus die **Nutzbarkeit mit stark eingeschränkten Ressourcen** relevant. Aus diesem Grund muss das Framework möglichst wenig Flash- und RAM-Overhead sowie geringe Hard- und Softwareanforderungen vorweisen - zur Erfüllung dieser Anforderung muss das Framework mit einem einfachen Checker auf einem ESP8266 mit 1 MiB Flash und unter 100 KiB RAM funktionieren, sowie ebenfalls auf einem regulären Linux-System.

A4 Um zertifizierende Algorithmen in einem Netzwerk aus mehreren Systemen nutzen zu können, ist es zudem notwendig, dass **alle für die Nutzung von Algorithmen relevanten Daten in einem Binär- oder Textprotokoll serialisierbar sind.** Da das System, auf dem der Algorithmus ausgeführt wurde, bei zertifizierenden Algorithmen keine Rolle spielt, und damit Netzwerkanwendungen häufig zu erwarten sind, sollte dies Teil des Frameworks sein. Ebenfalls lässt sich damit später einfacher eine Kompatibilität mit zusätzlichen Programmiersprachen herstellen.

A5 Aus demselben Grund ist ein **geringe Latenz bei der Nutzung über eine Netzwerkschnittstelle** wichtig. Die Bereitstellung des dafür notwendigen Servers selbst soll jedoch nicht Teil des Frameworks sein, da dies stark von der eingesetzten Hardware und sonstigen Umgebung abhängt - aus diesem Grund wird diese Anforderung lediglich durch die Übertragung per UART verifiziert, die maximale Zeit für die vollständige Antwort mit Überprüfung beträgt dabei 25 Millisekunden.

A6 Für komplexere Algorithmen ist außerdem (insbesondere für das Witness) die **effiziente Unterstützung für komplexe Datentypen** (mindestens Structs, Maps und Arrays) notwendig, wodurch unter anderem ein Graph als Zeuge für den `IsBipartite`-Algorithmus abbildbar sein muss. Langfristig könnten so auch beispielsweise Algorithmen zur Bildverarbeitung zertifizierend umgesetzt werden.

4.2 Erster Evaluationsschritt

Im ersten Evaluationsschritt (Iteration 1) wird ein grundlegendes Framework in reinem C implementiert, das ausschließlich anderen C-Code ausführen kann. Für die Kompilierung wird das System von RIOT OS [18] genutzt, damit kann das Ergebnis einfach auf verschiedenen Hardwareplattformen getestet werden.

4.2.1 Designüberlegungen

Für die Entwicklung wurden anhand der Anforderungen die folgenden Designentscheidungen getroffen.

Programmiersprache: C

Die Wahl auf C als Programmiersprache fällt vor allem aufgrund von Punkt A1 und Punkt A3: Projekte können unkompliziert speichersparend und performant entwickelt werden und mit vielen Hardwareplattformen kompatibel gemacht werden. Zudem besteht gleich auf mehreren Arten eine gute Kompatibilität zu anderen Programmiersprachen: etwa über Shared Libraries (z. B. Go, Rust), viele transpilierte Sprachen (z. B. Matlab) sowie die Verfügbarkeit von Interpretern zur Einbettung in C-Programme (z. B. Lua, WebAssembly).

Es wurden darüber hinaus mehrere modernere Programmiersprachen evaluiert, darunter Rust und Go. Während es hier ebenfalls eine gute Interoperabilität mit C und somit effektiv die gleichen Möglichkeiten zur Einbettung anderer Sprachen gibt, führt hier einerseits oft dennoch ein Umweg über C (womit sich die Zahl der zu verstehenden Konzepte und Werkzeuge mindestens verdoppelt), andererseits muss es dann zusätzlich eine Unterstützung der Hardware durch die entsprechende Sprache geben, was besonders bei neuen oder unbekannten Hardwareplattformen nicht immer der Fall ist beziehungsweise sich generell (im Gegensatz zur Unterstützung von C) von Hersteller zu Hersteller stark unterscheidet.

Unter C ist es beispielsweise mit RIOT, welches später auch bei der Evaluation genutzt wird, möglich, ein Programm für viele Hardwareplattformen gleichzeitig zu entwickeln, und mit Modulen auch viele bestehende Bibliotheken zu nutzen. Dazu kommt die Kompatibilität von C mit so gut wie jeder Hardware-Plattform über die vom Hersteller zur Verfügung gestellte Toolchain.

Punkt A2 soll erfüllt werden, indem bestimmte Code-Teile mit Compiler-Makros von der Kompilierung ausgeschlossen werden, dies ist mit C problemlos umsetzbar.

Aufrufmodell: Request/Response

Als Grundmodell für den Aufruf der Algorithmen und Checker wird ein Request/Response-Modell, wie es aus Client-Server-Anwendungen bekannt ist, angenommen. So kann ein Aufruf fast eins zu eins über ein Netzwerk umgesetzt werden, und alle notwendigen Daten können als ein Payload in Form von Binärdaten exakt so serialisiert werden, wie sie im Rahmen des Frameworks auch genutzt werden.

Eine weitere Option wäre ein RPC-orientiertes Modell, das auch weitere Aufruftypen wie Anfragen ohne oder mit mehreren Antworten unterstützt - dies wird bei synchronen zertifizierenden Algorithmen jedoch nicht benötigt, womit es fast identisch zum Request/Response-Modell genutzt werden würde. Auch eine Message Queue wäre denkbar, bei der Anfrage und Antwort vollständig unabhängig voneinander sind - somit kann etwa in einem Netzwerk ein System (oder auch mehrere) für die Ausführung gewählt werden, nachdem die Anfrage gestellt wurde. Die Wahl fällt jedoch auf das Request/Response-Modell, da dieses grundsätzlich leichter als andersherum auf diese und ähnliche Konzepte erweitert werden kann.

Als grundsätzlich anderer Ansatz wäre auch ein Header-only-Framework bei C möglich - so könnte man effektiv ein Framework komplett ohne Overhead umsetzen, hat jedoch den Nachteil, dass dieses auch ausschließlich mit C funktioniert, und die Serialisierung die Datenstrukturen nicht direkt abbilden kann.

Serialisierung: CBOR

Um für Punkt A4 die Anfrage- und Antwortdaten zu serialisieren, soll das Binärprotokoll „Concise Binary Object Representation“ (CBOR) nach [19] genutzt werden, da dieses Format gleichzeitig sehr kompakt ist, und es viele Bibliotheken gibt, die für eingebettete Systeme optimiert sind.

Alternativen wären beispielsweise JSON als Textprotokoll (dadurch mit deutlich größeren zu übertragenden Datenmengen) sowie Protocol Buffers als Binärprotokoll. Der konzeptuelle Unterschied zwischen CBOR und Protocol Buffers ist gering, CBOR wurde vorrangig aufgrund der besseren Integration mit RIOT OS gewählt.

Framework-Struktur und API

Die API beschreibt die Methoden und andere Schnittstellen, die bei der Nutzung des Frameworks direkt verwendet werden. Das Framework selbst teilt sich in die API zur Implementierung von Algorithmen und Checkern, sowie die API zur Nutzung der Algorithmen. Aus diesem Grund wird eine lokale „Registry“ eingeführt, für die sich Algorithmen registrieren können. Da eine Anfrage die ID des Algorithmus enthält, kann diese genutzt werden, um die Implementierung des Algorithmus auszuwählen. Somit ist die genaue Implementierung bei der Nutzung irrelevant und sogar austauschbar.

Eine einmalige ID wird für jeden Algorithmus und für jede Anfrage genutzt, um Verwechslungen zu vermeiden.

Ein Algorithmus besteht dabei immer aus seiner Implementierung (dem Executor) sowie dem Checker. Dazu kommen Funktionen zur Serialisierung und Deserialisierung in CBOR.

4.2.2 Funktionsweise & Implementierung

Das Projekt besteht aus den Bestandteilen `libceral` (dem eigentlichen Framework) sowie dem Algorithmus (`digitsum` oder alternativ `digitsum_matlab` als ein durch Matlab in C übersetzte Bibliothek) und einem Beispielprogramm als `main.c`.

Die `libceral` besteht aus folgenden Teilen:

- **Request & Response**: Datenmodelle für die Parameter (Request Params) sowie dem Paar aus Rückgabewert (Value) und Witness.
- **Encoding & Decoding**: Hilfsmethoden für die CBOR-Repräsentation von Request & Response, die zur deren Übertragung über ein Netzwerk genutzt werden können.
- **Algorithm Registry**: Eine Map aus Implementierungen von Algorithmen & Checkern, die über eine ID (hier eine Webadresse als String) abgerufen werden können.
- **Execution Wrappers**: Hilfsmethoden, um anhand eines Requests den dazugehörigen Algorithmus bzw. dessen Checker aufzurufen.

Damit ein Algorithmus mit `libceral` genutzt werden kann, benötigt er folgende Felder & Methoden:

- `algorithm_id`: Die ID des Algorithmus.

- `(encode|decode)_(params|value|witness)`: Methoden für die CBOR-(De)Kodierung von Parametern, Rückgabewert und Zeugen.
- `close_(request|response)`: Hilfsmethoden für das Speichermanagement der Request/Response-Objekten; sollten den gesamten Speicher von `params` beziehungsweise `value` und `witness` freigeben.
- `checker`: Überprüft die Korrektheit des Ergebnisses mit der Checker-Funktion.
- `executor`: Führt den eigentlichen Algorithmus aus.

Das CBOR-Format, in dem Request oder Response ausgetauscht werden, ist in Abbildung 4.1 ersichtlich. Die Struktur auf Grundlage eines Arrays wurde gewählt, da die Algorithmus-ID überprüft werden muss, bevor der Parser des Algorithmus die übermittelten Werte verarbeitet; theoretisch wäre dafür auch eine Map geeignet, da hier durchaus auch Bedingungen für die Feldreihenfolge genutzt werden können [19, Abschnitt 5.6], aufgrund der leichter ersichtlichen Semantik und einfacheren Konvertierbarkeit in andere Formate wie JSON wird jedoch dennoch ein Array als grundlegende Struktur genutzt.

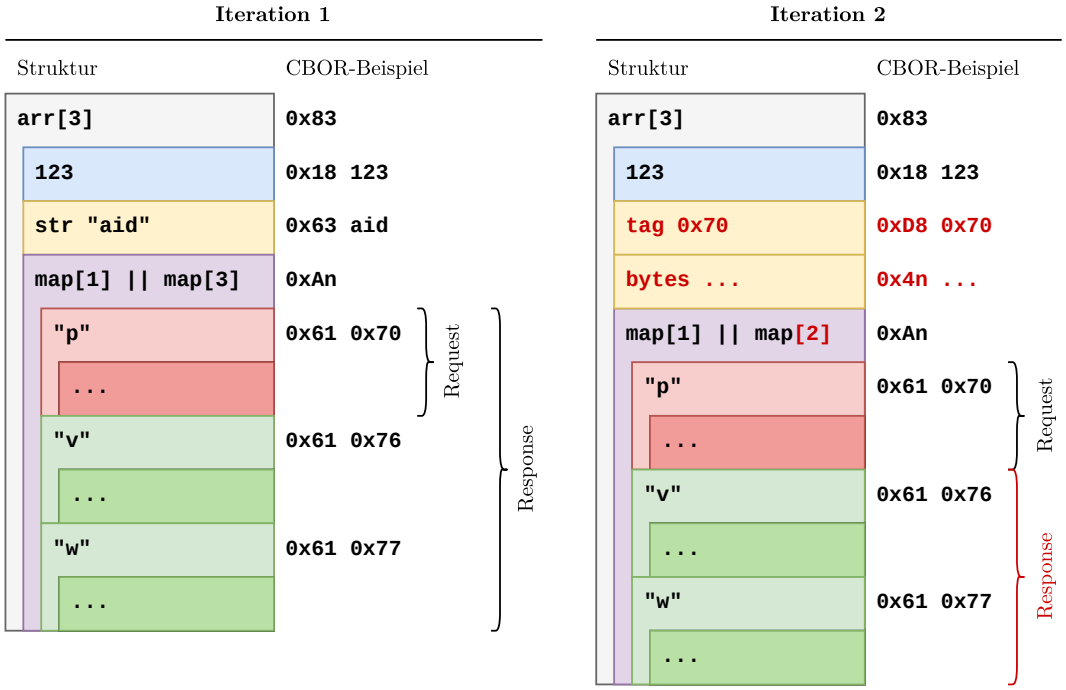


Abbildung 4.1: Aufbau des CBOR-Schemas für Iteration 1 und 2 im Vergleich

Im CBOR-Modell kommt eine Map zur einfacheren Analyse der Daten zum Einsatz; dabei steht `p` für die „params“ der Anfrage, `v` for den Rückgabewert „value“ und `w` für den „witness“. Da ein Checker immer auch die Anfrage benötigt, ist beides in einem gemeinsamen Datentyp zusammengefasst, `v` und `w` sind hier nur bei der Response erforderlich, ein Request nutzt stattdessen eine `map[1]` nur mit dem Feld `p`.

Um einen Algorithmus schließlich auszuführen, sowie das Ergebnis zu überprüfen, muss nun

der in Abbildung 4.2 dargestellte Prozess durchlaufen werden; dies wird beispielsweise auch in der Datei `main.c` implementiert.

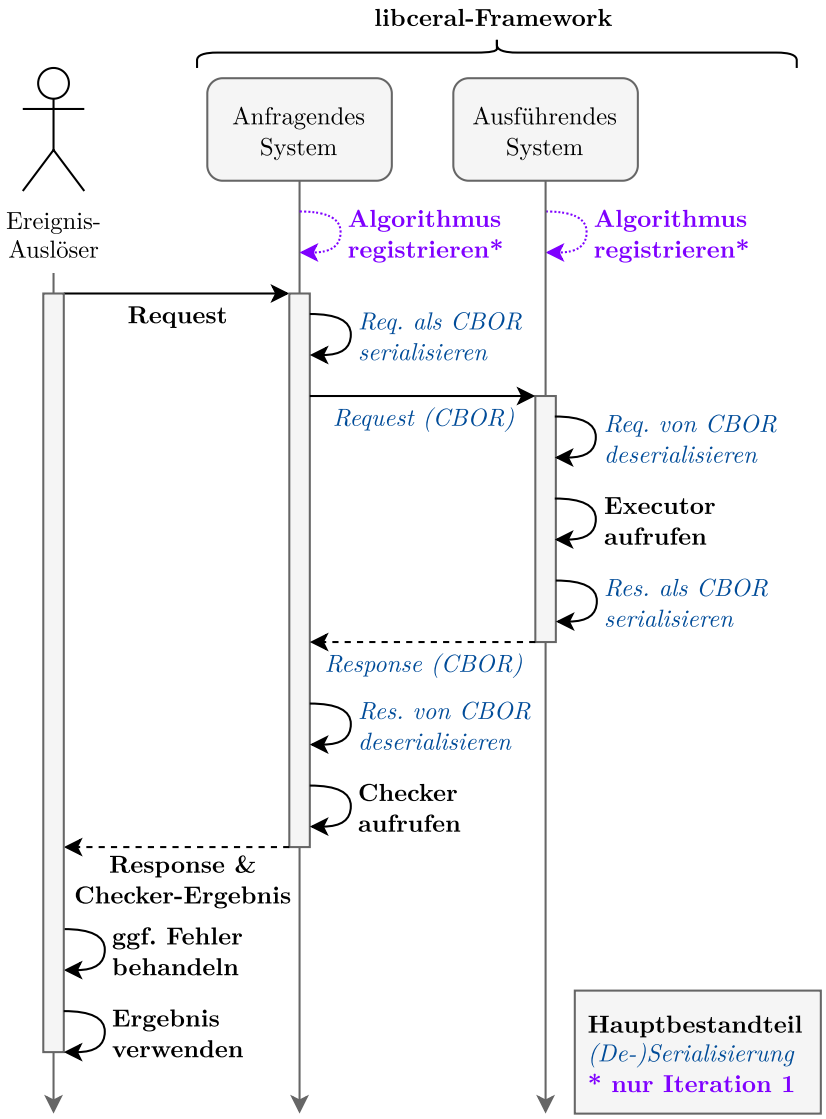


Abbildung 4.2: Schematischer Ablauf einer Berechnung mit dem libceral-Framework, mit Vergleich der zwei Iterationen. Die Datenübertragung zwischen anfragendem zum ausführendem System ist nicht Teil des Frameworks; es kann sich dabei auch um dasselbe System handeln - in diesem Fall ist die (De-)Serialisierung nicht notwendig.

4.2.3 Evaluation & Anforderungsermittlung

Als erstes Einsatzbeispiel dient zuerst ein sehr einfacher zertifizierender Algorithmus zur Berechnung einer Quersumme - der Witness ist dabei ein Array der einzelnen Ziffern. Offensichtlich ist dieses Beispiel eher zur Veranschaulichung der Nutzung geeignet, als dass es als zertifizierender Algorithmus wirklich sinnvoll wäre; an so einem einfachen Beispiel kann

man jedoch gut abschätzen, wie viel Mehraufwand die Nutzung des Frameworks bedeutet und ab wann sie sich lohnt - so verbraucht der Algorithmus mit dem Framework (gegenüber einer einfachen C-Funktion) wie in Abbildung 4.3 und Abbildung 4.4 ersichtlich 6 zusätzliche Codezeilen (4 bei der Implementierung und 2 bei der Nutzung des Algorithmus) sowie sehr viel mehr CPU-Instruktionen (bei nur leicht komplexeren Algorithmen sowie bei mehreren Durchläufen nähern sich diese Zahlen allerdings deutlich an), ca. 200 Byte mehr maximalen Arbeitsspeicher und ca. 1,8 KiB zusätzlichen Flash-Speicher.

Die Serialisierung von CBOR hat dabei die größten Auswirkungen auf die Performance, und sollte darum nur genutzt werden, wenn der Overhead durch die Netzwerkübertragung größer ist. Ermittelt werden konnte dies nur anhand von CPU-Instruktionen und Dateigröße, da aufgrund eines Fehlers bei der Ausführung mit Valgrind keine Werte für die Speichernutzung bei Iteration 1 gemessen werden konnten.

Anhand eines etwas komplexeren Algorithmus wie IsBipartite wird schon eher das Potenzial des Frameworks vergleichbar: der CPU-Overhead beträgt hier nur knapp 1 % bei der ersten beziehungsweise 4 % bei weiteren Ausführungen, während sich allerdings der maximal genutzte Arbeitsspeicher - vermutlich aufgrund der effizienten Nutzung von Datentypen im Algorithmus selbst - um fast 250 % erhöht, ähnlich sieht es bei der Dateigröße mit einer Erhöhung um 54 % aus. Der Mehraufwand an Codezeilen für Implementierung und Nutzung ist generell recht gering, wird jedoch durch die Nutzung von CBOR bei komplexen Datentypen in diesem Fall immerhin noch einmal verdoppelt.

Diese in Abbildung 4.3 und Abbildung 4.4 dargestellten Zahlen wurden dabei folgendermaßen ermittelt - grundsätzlich sind dabei bei allen Zahlen geringere Werte besser:

- Code-Statements: Es erfolgt eine manuelle Zählung der notwendigen Semikolon-Zeichen im Code - relevant für die Benutzerfreundlichkeit.
- Kompilierte Dateigröße: Angegeben ist die Dateigröße der kompilierten Binärdatei in Bytes - relevant für Hardwarebeschränkungen.
- CPU-Instruktionen: Gemessen wird mit dem Linux-Syscall `perf_event_open` [20] - relevant ist der Wert für die Ausführungsgeschwindigkeit. Der erste Aufruf benötigt vermutlich aufgrund von statischen Deklarationen mehr CPU-Instruktionen, weshalb dieser Wert zwei Mal gemessen wird, einmal für den ersten und einmal für weitere Aufrufe. Dieser Wert ist exakt, solange die Messung auf einem identischen System ausgeführt wird.
- Arbeitsspeichernutzung: Genutzt wird Valgrind mit Massif als Heap- und Stack-Profiler [21], als Kennziffern werden die maximale Speicher-Belegung zu einem bestimmten Zeitpunkt (relevant für Hardwarebeschränkungen) sowie die Zahl der insgesamt allozierten bzw. deallozierten Bytes (wird von Massif als Zeiteinheit behandelt; relevant für die Ausführungsgeschwindigkeit). Der Wert ist je analysierte Programmdatei exakt.

Mit der 1. Iteration konnten folgende Ziele erfüllt beziehungsweise verifiziert werden:

Punkt A1 ist durch die Wahl von C als Programmiersprache grundsätzlich gegeben, erfolgreich getestet wurde die Algorithmen-Implementierung in Matlab mit einer Transpilierung in C; die Nutzung der Algorithmen wurde jedoch noch nicht getestet.

Punkt A2 ist dadurch erfüllt, dass C mit Makros zur Compile-Zeit gesteuert werden

kann. Der Executor kann also beispielsweise mit `#if defined(CERAL_EXECUTOR) || defined(DIGITSUM_CERAL_EXECUTOR)` umschlossen werden, sodass, wenn `CERAL_EXECUTOR` nicht definiert ist, nur der Checker kompiliert wird. Durch die zwei Makros ist es auch möglich, nur für einzelne Algorithmen den Executor mitzukompilieren.
Für detailliertere Benchmarks gibt es zudem das nicht näher dokumentierte Makro `CERAL_DISABLE_CBOR` - ist es definiert (und vom Algorithmus ebenfalls unterstützt), so werden die CBOR-Teile nicht mitkompiliert.

Punkt A3 ist erfüllt, da die Beispiele erfolgreich auf einem ESP8266 sowie auf einem Linux-System ausgeführt werden können.

Punkt A4 ist durch die (De-)Serialisierbarkeit mit CBOR gegeben, diese macht es jedoch mit dem Grad der Abstraktion des Frameworks nicht möglich, die klassische statische Speicherverwaltung von C zu nutzen.

Punkt A5 wurde bisher nicht direkt getestet, zur Übertragung wird stattdessen ein einfaches UART-Protokoll genutzt. Grundsätzlich ist jedoch jedes Protokoll nutzbar, welches binäre Daten übertragen kann.

Punkt A6 wurde mit dem IsBipartite-Algorithmus getestet, grundsätzlich können jedoch alle in C repräsentierbaren Datentypen genutzt werden - beim Einsatz von Pointer-basierten Datenstrukturen ist jedoch eine Serialisierung mit CBOR schwieriger, ein Graph sollte darum beispielsweise besser mit zwei Arrays aus Knoten und Kanten dargestellt werden, bei denen der Array-Index statt der Speicheradresse für Referenzen genutzt wird.

Folgende Probleme sind bei der Implementierung, Nutzung und Evaluation aufgefallen:

- Für die Nutzung verschiedener Algorithmen mit CBOR ist dynamische Speicherverwaltung notwendig, Memory-Leaks sind somit deutlich wahrscheinlicher.
- Die Identifizierung von Algorithmen durch eine URL macht die serialisierten Daten unnötig groß, eventuell kann hier ein kleinerer Identifier wie beispielsweise eine OID (Object Identifier nach ITU-T) genutzt werden.
- Die Algorithmen-Registry abstrahiert vieles vom Nutzer weg; so ist bei der Ausführung eines Requests aus dem reinen Code unklar, welcher Algorithmus dafür verwendet wird.
- Die grundlegende Struktur der API des Frameworks funktioniert, die Nutzung hängt jedoch noch immer stark von der Projektumgebung ab, vor allem bei der Initialisierung von algorithmenspezifischen Datentypen, und es gibt teilweise Inkonsistenzen bei Begriffen. Außerdem könnten mit Hilfsfunktionen noch einige Codezeilen eingespart werden, sowie durch eine einfache Restrukturierung auch eine Nutzung direkt mit einer einzigen `#include`-Direktive für in Benchmark- und Testprojekten ermöglicht werden, statt einen Build als RIOT-Modul oder Shared Library notwendig zu machen.

Für die zweite Iteration müssen darum folgende zusätzliche Änderungen umgesetzt werden:

A7 Explizite Speicherverwaltung ohne versteckten (De-)Allokationen.

A8 Identifizierung von Algorithmen mit einer kompakteren ID.

A9 Explizite Angabe der zu nutzenden Implementierung eines Algorithmus.

A10 Konsistente Namensgebung und Vermeidung von Codezeilen, indem mehrere Hilfsfunktionen für die Nutzung und Makros für die Implementierung von Algorithmen

zur Verfügung gestellt werden.

Ebenfalls wurden Punkt A5 und Punkt A1 noch nicht voll erfüllt beziehungsweise getestet und müssen darum reevaluiert werden.

Die Grundlagen werden durch die 1. Iteration recht gut abgedeckt: das Framework ist grundsätzlich für den Einsatzzweck tauglich und kann so wie es ist verbessert werden - die meisten Änderungen sollten recht unproblematisch angewendet werden können. Einzig die Notwendigkeit der dynamischen Speicherverwaltung (die besonders in RIOT keine Best Practice ist) kann mit dem für die (De-)Serialisierung gewählten Ansatz vermutlich nicht umgangen werden, da unterschiedliche Algorithmen stark unterschiedliche Datenstrukturen und damit Anforderungen an den Speicheraufbau haben.

4.3 Zweiter Evaluationsschritt

Für die zweite Iteration des Frameworks werden neben den in der ersten Iteration festgestellten notwendigen Änderungen auch viele Erleichterungen für den Nutzer umgesetzt. Ziel ist hier insbesondere, die API für die nächste Zeit festzusetzen, sodass alle üblichen Aufgaben damit einfach umgesetzt werden können und das Framework so wenig wie möglich im Weg steht. Alle zur Nutzung notwendigen Informationen finden sich in Abschnitt 5.3 sowie in der im Code enthaltenen Dokumentation.

4.3.1 Designüberlegungen

Dynamische Speicherverwaltung

Die wichtigste zu evaluierende Änderung nach der letzten Iteration betrifft dabei die Speicherverwaltung - eine statische Speicherverwaltung wäre aufgrund der RIOT-Best-Practices[22] wünschenswert, würde aber große Teile des Frameworks redundant machen, da alle Variablen sowohl bei der Vorbereitung eines Requests als auch bei der Deserialisierung manuell deklariert werden müssten. Gerade bei komplexen Algorithmen, die mit CBOR-serialisierten Parametern aufgerufen werden sollen, wären hier deutlich mehr Zeilen Code pro Aufruf notwendig, was den Nutzen des Frameworks gegenüber einer manuellen Implementierung verringert. Auch die Nutzung variabler Datentypen wie Strings in übergebenen Werten würde so schwieriger bis unmöglich, vor allem auch bei der CBOR-Kodierung.

Eventuell wäre es für die Umsetzung einer solchen flexible statischen Speicherverwaltung in C (die den Codeaufwand bei der Nutzung gering hält) eine sinnvolle Möglichkeit, ein auf Compiler-Makros basierendes Framework zu entwerfen, wobei jedoch die Kompatibilität zu anderen Programmiersprachen besonders als geteilte Library nicht mehr gewährleistet werden könnte. Während Compiler-Makros durchaus bereits in der ersten Iteration genutzt werden, sind diese nicht zwingend zur Nutzung des Frameworks notwendig und kommen nicht bei der Nutzung, sondern vor allem auf die Implementierung von Algorithmen zum Einsatz.

Die Alternative dazu stellt wie in Iteration 1 genutzt die dynamische Speicherverwaltung mit `malloc` und `free` dar, welche zwei große Nachteile mit sich bringt: zum einen fehlt

bei RIOT auf einigen wenigen Hardwareplattformen[22] eine Implementierung vollständig; diese Plattformen würden also gar nicht unterstützt werden. Der andere Nachteil liegt in der geringeren Automatisierung durch den Compiler - alle einmal allozierten Ressourcen müssen später wieder freigegeben werden; dafür ist es essentiell, dass der Nutzer weiß, wann welche Ressourcen alloziert werden - ist das nicht der Fall, entstehen Memory Leaks und damit die Gefahr, dass ein System unerwartet ausfällt.

Die Vorteile dynamischer Speicherverwaltung überwiegen jedoch, weshalb diese weiterhin für das Framework genutzt werden soll. Um das Risiko des zweiten Nachteils etwas zu verringern, werden die Begriffe in den Methodenaufrufen explizit festgelegt - diese nutzen also immer den Präfix `new_` oder `close_`, um ersichtlich zu machen, welche dynamisch allozierte Ressourcen später unbedingt wieder freigegeben werden müssen.

„Skelett“ mit OID als globale ID für Algorithmen

Dadurch, dass Algorithmen nicht nur an eine Implementierung geknüpft sind, können Datentypen und Serialisierung sowie Checker wiederverwendet werden - die Checker sind grundsätzlich bereits eine einfach wiederverwendbare Funktion; die Abgekoppelten Datentypen mit Serialisierung werden im Folgenden „Skelett“ eines Algorithmus genannt.

Ein solches muss global eindeutig identifiziert werden können - in Iteration 1 wurde dafür ein URL-String genutzt, der jedoch zweierlei Probleme mit sich bringt: einerseits verbraucht er bei der Übertragung nicht unerhebliche Datenmengen, andererseits ist er nicht ausreichend standardisiert - wer darf welche Domain nutzen, was passiert wenn Domains nicht mehr genutzt werden, wie werden Versionen spezifiziert, und so weiter. Abhilfe kann hier der von der ITU-T spezifizierte OID-Standard schaffen [23]: Mit diesem gibt es eine zentral verwaltete Registry, welche dauerhafte hierarchische IDs nutzt, welche auch binär enkodiert werden. Mit [24] steht sogar ein Standard für die Enkodierung in CBOR zur Verfügung.

4.3.2 Funktionsweise & Implementierung

Die weiteren vorgeschlagenen Änderungen wurden wie in der vorhergehenden Evaluation umgesetzt. Insgesamt gibt es in der zweiten Iteration damit folgende Änderungen an der API:

- Fester Grundsatz, dass nur in Methoden mit dem Präfix `new_` neue dynamische Ressourcen alloziert werden dürfen, und diese auch nur in Methoden mit dem Präfix `close_` wieder freigegeben werden dürfen.
- Wechsel von URLs auf OIDs als Algorithmus-ID: so müssen weniger Daten übertragen werden, die Algorithmen sind jedoch noch immer eindeutig zu identifizieren. Auch die Einführung globaler Registries ist damit möglich, wie beispielsweise die in Abschnitt 5.4 beschriebene Registry.
- Wegfall der lokalen Algorithmen-Registry: stattdessen muss die zu nutzende Implementierung nun jeweils bei den Methodenaufrufen mit übergeben werden. So muss ein Algorithmus einerseits nicht erst registriert werden, andererseits ist es auch möglich unterschiedliche Implementierungen eines Algorithmus miteinander zu vergleichen.
- Aufsplittung von Request und Response: Eine Response enthält nun nicht mehr auch

den Request, was die übertragene Datenmenge gerade für Algorithmen mit größeren Datenstrukturen (Graphen, Bilder oder ähnliches) verringern soll. Entsprechend ändert sich auch die Definition des CBOR-Protokolls, sodass dieses nun *entweder* den Map-Parameter `"p"` *oder* die beiden Map-Parameter `"v"` und `"w"` enthält.

- Einführung einer Konvention für die Extraktion von Parametern, Rückgabewert und Witness mit `ALGORITHM_params(req)`, `ALGORITHM_value(res)` und `ALGORITHM_witness(res)`
- Auslagerung der Framework-internen Methoden eines Algorithmus in ein eigenes Skelett-Objekt: dadurch wird einerseits die Syntaxvervollständigung kürzer und dadurch sinnvoller, andererseits wird es dadurch möglich, Code für beispielsweise die (De-)Serialisierung oder den kompletten Checker in unterschiedlichen Implementierungen desselben Algorithmus wiederzuverwenden.
- Es gibt nun einige zusätzliche Compiler-Makros für die Definition von Algorithmen, darunter `CERAL_ALGORITHM` zur Initialisierung des `CeralAlgorithm`-Structs, sowie `CERAL_HANDLE_CBOR_ERROR` zur einfacheren Fehlerbehandlung von CBOR-Fehlern.

Es sind zudem noch einige Beispiele hinzugekommen beziehungsweise verbessert, darunter auch die Implementierung des `IsBipartite` Algorithmus in Go, sowie etwas besser strukturiertere Beispiele für die separate Betrachtung von (De-)Serialisierung und der Nutzung mit RIOT.

Auch Unit-Tests für den IsBipartite-Algorithmus wurden hinzugefügt, genutzt wird dabei wie bei RIOT auch das EmbUnit-Framework[2].

4.3.3 Evaluation & Anforderungsermittlung

Die Nutzung in anderen Programmiersprachen (Punkt A1) wurde erneut anhand von Go evaluiert, nun auch mit IsBipartite für die Implementierung eines Algorithmus. Dies funktioniert unproblematisch, das Framework kann fast wie in C genutzt werden, es muss für die Verwendung dieser Algorithmen jedoch eine Shared Library genutzt werden. Viele andere Programmiersprachen (darunter Java, Rust, u.v.m.) unterstützen einen ähnlichen Ansatz, alternativ wurden zu C transpilierbare Sprachen wie Matlab bereits in der 1. Evaluation betrachtet. Es gibt allerdings auch einige Sprachen, bei der die Lösung in die andere Richtung führt: mit interpretierten Sprachen wie Lua oder JavaScript beispielsweise ist es möglich, ein C-Programm als Einstiegspunkt zu nutzen, welches die notwendigen Funktionen dem Skript über den Interpreter direkt zur Verfügung stellt. Dies sollte äquivalent zur Nutzung in reinem C funktionieren, es ist aber natürlich zu beachten dass es in den meisten fertigen Lua- und JavaScript-Umgebungen nicht möglich ist, das Framework so wie es ist zu nutzen.

Die Zeit für eine komplette Ausführung des DigitSum-Algorithmus auf einem ESP8266 von einer X86-Maschine aus (nicht realitätsnah, aber andersherum sollte die Zeit eher noch kleiner sein) über ein einfaches UART-Protokoll (Baud-Rate 115200) beträgt nach Tabelle 4.1 ca. 10 Millisekunden, womit nun auch die Erfüllung von Punkt A5 bestätigt ist.

Die Probleme mit der Speicherverwaltung (Punkt A7) wurden vorrangig durch bessere

[2] `https://sourceforge.net/projects/embunit/`, abgerufen 28. März 2022

Namenskonventionen mitigiert.

Punkt A8 ist durch die Nutzung von OIDs erfüllt, und wird gemeinsam mit Punkt A9 durch den Ansatz des Algorithmus-Skeletts gelöst: Dieses enthält nur die Spezifikation der Rahmenbedingungen, die eigentliche Implementierung ist davon unabhängig und muss darum jedes Mal angegeben werden.

Punkt A10 wurde unter anderem durch die Einführung von Konventionen für Datentypen-Hilfsfunktionen gelöst, darunter `new_myalgorithm_params(...)`.

Die Metriken, welche vor allem für Punkt A3 relevant sind, zeigen, dass Iteration 2 vor allem bei der CBOR-(De-)Serialisierung vorne liegt, wobei in diesem Fall jedoch etwas mehr Flash-Speicher verbraucht wird. Andere Werte ändern sich nur leicht: Vor allem verringern sich die gesamten (Re-)Allokationen, und für etwas weniger Codezeilen bei der Nutzung sind nun etwas mehr Codezeilen bei der Implementierung des Algorithmus notwendig.

Nächste Schritte

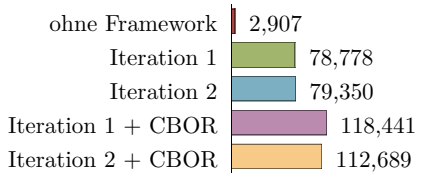
Als nächsten Schritt der Erweiterung des Frameworks ist die Bereitstellung eines generischen Serverprozesses für Inter-Prozess-Kommunikation auf Mehrprozesssystemen oder sogar für die Kommunikation über das Netzwerk denkbar, beispielsweise über gRPC, CoAP, MQTT oder HTTP. Damit wäre es möglich, Algorithmen auf beispielsweise Linuxsystemen über den Paketmanager zu installieren, und danach von beliebigen Programmiersprachen ais unter der Nutzung einer generischen Client-Library für eines dieser Protokolle sowie einer generischen Library für CBOR zu nutzen. Dies kann recht einfach auf Grundlage dieser Framework-Iteration umgesetzt werden, noch zu evaluieren ist hierbei jedoch, inwieweit es Sinn ergibt, auch den Checker über eine IPC-Schnittstelle anzusprechen - als zusätzlicher Point of Failure kann hier natürlich das Checker-Ergebnis manipuliert werden, es ist also mehr Vertrauen in das System notwendig als wenn der Checker direkt in derselben Anwendung läuft, die auch den Aufruf des Algorithmus auslöst.

Für eine einfache Integration mit beliebigen Netzwerkprotokollen ist hier auch ein Ansatz denkbar, bei dem auf eine zu einer Anfrage gehörigen Funktion gewartet wird - die Request-ID wird in einen Pool geschrieben, dann kann auf die dazu passende Response gewartet werden, unabhängig von der eingesetzten Netzwerkbibliothek. Insbesondere bei Interrupt-basierten Netzwerkbibliotheken ist dies vermutlich eine sinnvolle Ergänzung.

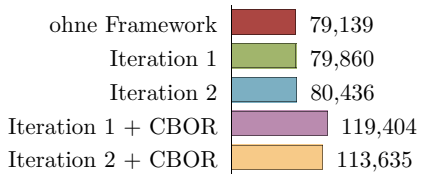
Ebenfalls könnte ein weiterer Schritt sein, das Framework auch auf eine Nutzung mit verteilten zertifizierenden Algorithmen nach Völlinger [3] hin zu erweitern. Durch die besonderen Anforderungen ist hier jedoch fraglich, ob dies überhaupt möglich ist, ohne die bisherigen Anforderungen zu verletzen: beispielsweise die Feststellung der Terminierung würde beim gegebenen Framework vermutlich einen nicht zu vernachlässigenden Overhead erzeugen, weshalb insbesondere Punkt A2 daraufhin angepasst werden muss, dass auch nicht-verteilte Algorithmen ohne den nur für verteilte Algorithmen notwendigen Teilen kompiliert werden können. Die einfachste Lösung wäre hier die Einführung eines zusätzlichen Frameworks (beispielsweise *libceral-distributed*), welches auf *libceral* aufbaut und es insbesondere für Serialisierung, Speicherverwaltung und grundlegende Datenstrukturen nutzt, selbst jedoch eine für verteilte Algorithmen angepasste API bietet.

(a) CPU-Instruktionen (erster Aufruf)

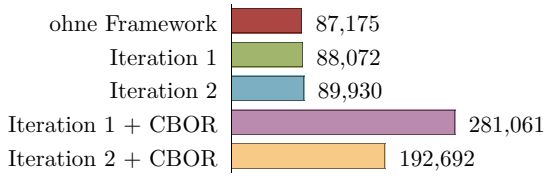
i. digitsum



ii. digitsum_matlab

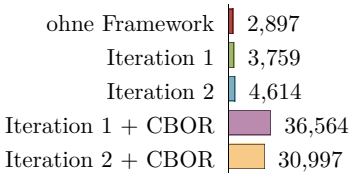


iii. isbipartite

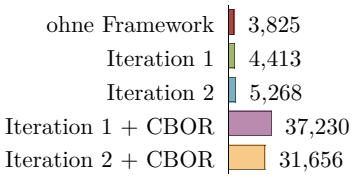


(b) CPU-Instruktionen (weitere Aufrufe)

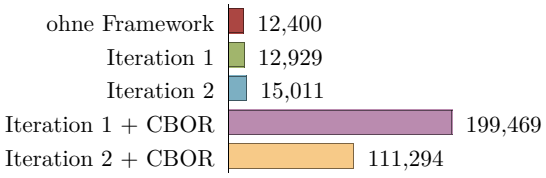
i. digitsum



ii. digitsum_matlab

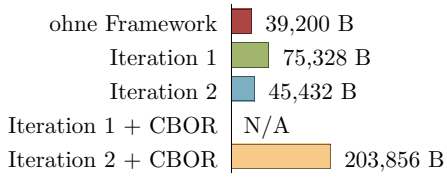


iii. isbipartite

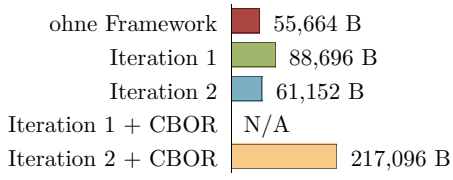


(c) Insgesamt (re)allozierter Speicher

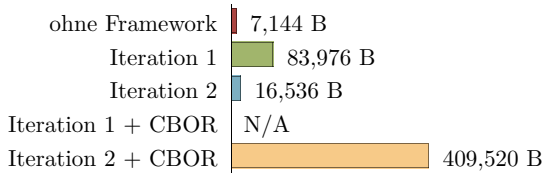
i. digitsum



ii. digitsum_matlab

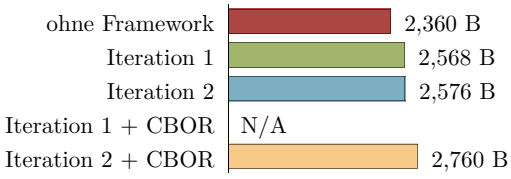


iii. isbipartite

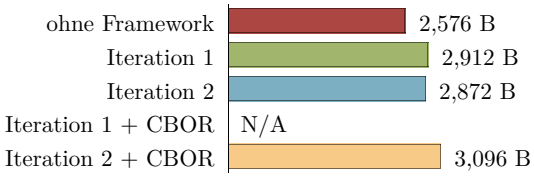


(d) Maximaler Speicher zu einem Zeitpunkt

i. digitsum



ii. digitsum_matlab



iii. isbipartite

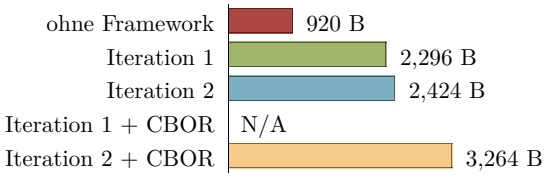
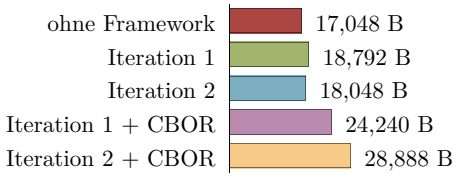


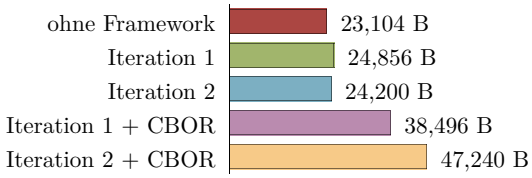
Abbildung 4.3: Benchmark-Ergebnisse für CPU und Speicher

(e) Kompilierte Dateigröße

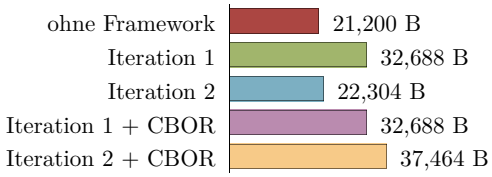
i. digitsum



ii. digitsum_matlab

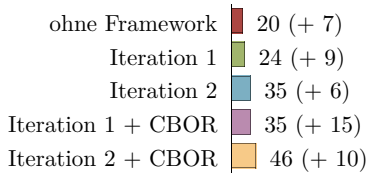


iii. isbipartite

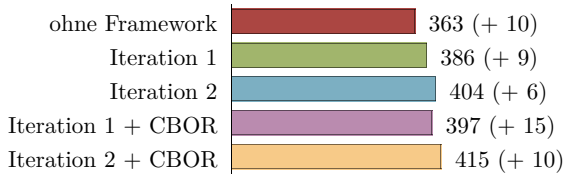


(f) Code-Statements (+ für Nutzung)

i. digitsum



ii. digitsum_matlab



iii. isbipartite

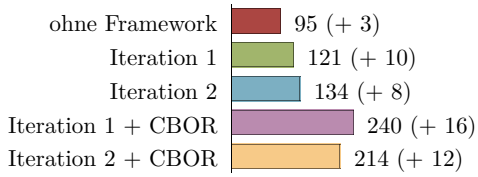


Abbildung 4.4: Benchmark-Ergebnisse für Dateigröße und Code-Statements

Requester

	Total	**Serializing**	**Sending**	**Waiting**	**Deserializing**	**Checking**	***Latency***
1	*10439 µs*	23 µs	48 µs	10302 µs	42 µs	24 µs	4329 µs
2	*9875 µs*	4 µs	60 µs	9795 µs	12 µs	4 µs	4321 µs
3	*10043 µs*	5 µs	53 µs	9964 µs	15 µs	6 µs	4402 µs
4	*9912 µs*	5 µs	79 µs	9811 µs	13 µs	4 µs	4332 µs
5	*9863 µs*	5 µs	72 µs	9765 µs	16 µs	5 µs	4307 µs
avg	***10026 µs***	**8 µs**	**62 µs**	**9927 µs**	**20 µs**	**9 µs**	**4338 µs**

Responder

	Total	**Deserializing**	**Execution**	**Serializing**	**Sending**
1	*1644 µs*	709 µs	520 µs	372 µs	43 µs
2	*1153 µs*	497 µs	402 µs	220 µs	34 µs
3	*1160 µs*	493 µs	409 µs	224 µs	34 µs
4	*1147 µs*	493 µs	400 µs	220 µs	34 µs
5	*1151 µs*	492 µs	400 µs	225 µs	34 µs
avg	***1251 µs***	**537 µs**	**426 µs**	**252 µs**	**36 µs**

Tabelle 4.1: Benchmark-Ergebnisse für Ausführungszeit (nur Iteration 2 mit Digitsum und ESP8266)

KAPITEL 5

Ergebnisse

Im vorherigen Kapitel wurde ein Software-Framework für zertifizierende Algorithmen entwickelt und evaluiert. In diesem Kapitel wird zunächste das Ergebnis der Evaluation kurz zusammengefasst, und daraufhin die Nutzung des Ergebnisses beschrieben, sowie die daraus abzuleitenden Best Practices ermittelt, die so allgemeingültig wie möglich für Frameworks für zertifizierende Algorithmen, aber auch für die unabhängige Implementierung solcher Algorithmen gelten sollen. Für bestimmte Best Practices wird noch beispielhaft auf eine mögliche Umsetzung eingegangen. Schließlich werden mit dem gewonnenen Wissen noch einmal detaillierter zertifizierende Algorithmen mit ihren Alternativen verglichen.

5.1 Erfüllung der Anforderungen

Die Anforderungen konnten größtenteils erfüllt werden - die Hauptziele können dabei zusammengefasst werden in drei Kategorien: a) Interoperabilität zwischen Programmiersprachen, b) Ressourceneffizienz mit Blick auf eingeschränkte eingebettete Systeme sowie c) Nutzerfreundlichkeit für Algorithmen-Ersteller und -Nutzer. Eines davon vollständig zu erfüllen bedeutet immer auch Kompromisse bei den anderen zu machen, weshalb die vorgestellte Lösung einen Kompromiss zwischen diesen Zielen darstellt.

Die Eignung für verschiedene Fälle kann wie folgt zusammengefasst werden: Sollen Berechnungen auf demselben System durchgeführt werden, ist der Aufwand relativ gering - bei vielen Programmiersprachen außerhalb von C gibt es jedoch mehr sprachenspezifische Möglichkeiten zur Fehlerbehandlung, welche die Nutzung derart vereinfachen könnten, dass es gar nicht direkt auffällt, dass ein Algorithmus zertifizierend ist. Für die Auslagerung von Berechnungen auf *ein* anderes System ist das Framework sehr gut geeignet: in diesem Fall lässt sich der Mehraufwand kaum vermeiden und ist mit dem Framework sogar geringer als bei einer individuellen Lösung. Soll eine Berechnung auf mehreren Systemen gemeinsam ausgeführt werden, ist zusätzliche Arbeit notwendig, da dies je nach Algorithmus variiert.

5.2 Best Practices

Bei den in Abschnitt 3.2 vorgestellten Implementierungen konnten sowohl einige Probleme als auch einige Herangehensweisen mit positiven Auswirkungen identifiziert werden. Dies lässt sich in den folgenden ersten Best Practices zusammenfassen, die nach jeder Iteration jeweils erneut evaluiert und ergänzt werden:

BP1 Die Datenstrukturen, die ein Algorithmus nutzt, sollten unabhängig von der Implementierung und Programmiersprache definiert sein, damit unterschiedliche Implementierungen eines Algorithmus getestet werden können.

BP2 Es sollte eine Programmiersprache gewählt werden, die von möglichst vielen Projekten auf möglichst vielfältigen Hardwareplattformen genutzt werden kann - Abhängigkeiten sollten weitestgehend vermieden werden, außer die Hardware-Unterstützung und Lizenz ist identisch.

BP3 Der Checker sollte formal verifizierbar sein, gleichzeitig jedoch so kompatibel wie möglich, was die Wahl der Programmiersprache erschwert - dies muss bei der Wahl des Verifikations-Tools berücksichtigt werden.

BP4 Die Einführung von Preconditions kann dabei helfen, Fehlerfälle von vorneherein zu vermeiden - dabei muss auch beispielsweise die Größe von Datentypen berücksichtigt werden, um Fehler durch Überläufe zu vermeiden. [15] Diese Preconditions müssen sowohl vom Algorithmus, als auch vom Checker überprüft werden, im Idealfall gibt bereits der Algorithmus einen Fehler zurück, und ein mit fehlerhaften Eingabewerten.

BP5 Der Checker sollte mindestens alle Fälle korrekt abdecken, für die der Algorithmus ein *falsches* Ergebnis zurückliefert. Beispielsweise der Maximum-Cardinality-Checker aus LEDA tut dies wohl nicht, da nicht überprüft wird, ob das Ergebnis überhaupt ein Untergraph der Eingabe ist [8, S. 268], wodurch gegebenenfalls ein gefälschtes Witness genutzt werden kann.

BP6 Der Checker sollte wenn möglich ohne Kenntnis der Implementierung des Algorithmus entwickelt werden - nur so ist er vollständig unabhängig von der Implementierung und man wird nicht so schnell durch Implementierungsfehler beeinflusst.

Nach der Evaluation der ersten Iteration in Unterabschnitt 4.2.3 kommt folgende neue Best Practice dazu:

BP7 Implementierungen sollten so geschrieben werden, dass der Ausführungspfad einfach nachvollziehbar ist. Dies schließt gegebenenfalls den Verzicht auf unnötige Abstraktionsschichten ein - jedem Abstraktionsschicht muss weiträumig vertraut werden können. (abgeleitet aus Punkt A10)

BP8 Besonders bei der Nutzung von Abstraktionsschichten ist zudem relevant, dass nur notwendige Komponenten im finalen Programm enthalten sein sollten, damit Ressourcen möglichst effizient genutzt werden. (abgeleitet aus Punkt A2)

Die zweite Evaluation setzt dabei vor allem die Best Practices vollständig um. Es wurden hier keine neuen Best Practices mehr gefunden, aber alle hier aufgezählten Best Practices können damit umgesetzt werden.

5.3 Nutzung des Frameworks

In diesem Abschnitt wird die Nutzung des *libceral*-Frameworks aus Nutzersicht erläutert, um die relevanten Teile der Spezifikation (siehe Abschnitt A.2) zu veranschaulichen und eine beispielhafte Grundlage zu schaffen, auf welche die Best Practices aus dem vorherigen Abschnitt leicht angewendet werden können.

5.3.1 Nutzung von Algorithmen und Checkern

Um einen zertifizierenden Algorithmus auszuführen und das Ergebnis zu überprüfen, sind mit dem Framework grundsätzlich die in Quellcode 5.1 dargestellten Schritte notwendig.

```
1  // 1. Laden der Library und des Algorithmus
2  #include <ceral.h>
3  #include "myalgorithm.c"
4  // ...
5  // 2. Erzeugen des Requests
6  CeralRequest = new_myalgorithm_request(x, y);
7  // 3. ggf. enkodieren des Requests als CBOR; Übertragung auf andere Node
8  uint8_t *req_enc = malloc(CERAL_CBOR_DEFAULT_BUFFER_SIZE);
9  size_t req_len = ceral_encode_request(&myalgorithm, req, req_enc,
       CERAL_CBOR_DEFAULT_BUFFER_SIZE);
10 // 4. ggf. dekodieren des Requests von CBOR
11 CeralRequest *exec_req = ceral_new_request_from_cbor(&myalgorithm, req_enc,
       req_len);
12 // 5. Allozieren der Response
13 CeralResponse *exec_res = ceral_new_response(exec_req);
14 // 6. Ausführen des Algorithmus
15 CeralError err = ceral_execute(&myalgorithm, exec_req, exec_res);
16 // 7. ggf. enkodieren der Response als CBOR; Übertragung auf urspr. Node
17 close_ceral_request(exec_req);
18 uint8_t *res_enc = malloc(CERAL_CBOR_DEFAULT_BUFFER_SIZE);
19 size_t res_len = ceral_encode_response(&myalgorithm, exec_res, res_enc,
       CERAL_CBOR_DEFAULT_BUFFER_SIZE);
20 close_ceral_response(exec_res);
21 // 8. ggf. dekodieren der Response von CBOR
22 CeralResponse *res = ceral_new_response_from_cbor(&myalgorithm, res_enc,
       res_len);
23 // 9. Ausführen des Checkers
24 bool chk = ceral_check(&myalgorithm, req, res);
25 close_ceral_request(req);
26 // 10. Fehler behandeln
27 if (!chk) {
28     // ...
29     return 1;
30 }
31 // 11. Ergebnisse weiterverarbeiten
32 MyAlgorithmValue *x = myalgorithm_value(res);
33 // ...
34 // 12. Resourcen wieder schließen
35 close_ceral_response(res);
```

Quellcode 5.1: Beispiel für die Nutzung eines zertifizierenden Algorithmus mit *libceral*

Dies ist dabei der vollständige notwendige Code inklusive der Serialisierung und Deserialisierung als CBOR; wenn der Algorithmus auf dem selben System ausgeführt werden soll, fallen die Schrite 3, 4, 7 und 8 weg.

Implementierungen von Algorithmen können auch weitere Hilfsfunktionen einführen, wie beispielsweise zur einfacheren Arbeit mit den genutzten Datentypen.

5.3.2 Implementierung eines Algorithmus-Skeletts

Ein Algorithmus ist grundsätzlich durch eine Instanz des `CeralAlgorithm`-Structs definiert. Dieser muss immer eine Algorithmus-ID in Form einer CBOR-enkodierten OID enthalten [24], welche beispielsweise wie in Abschnitt 5.4 beschrieben beantragt werden kann. Dazu kommt die Implementierung des Algorithmus selbst, der Checker sowie die (De-)Kodierungsmethoden. Dazu sollte eine Initialisierungsfunktion für Requests mit der Namenskonvention `new_myalgorithm_request` existieren, die für einen Request notwendige Werte entgegennimmt und entsprechend Ressourcen für einen Request alloziert. In jedem Fall müssen zudem die Methoden `close_request` sowie `close_response` festgelegt sein, die die in der Initialisierungsfunktion im Request bzw. in der Ausführungsfunktion in der Response allozierten Speicherbereiche wieder freigeben.

Für einen Algorithmus oder Checker ist also zuerst eine Grundlage aus Datentypen sowie deren Speicherverwaltung und (De-)Serialisierung notwendig, diese wird im Rahmen des *libceral*-Frameworks als Skelett des Algorithmus bezeichnet - dieses kann grundsätzlich auch teilweise von mehreren Algorithmen wiederverwendet werden, wenn diese die gleichen Datenstrukturen nutzen. Die Information was der Algorithmus tut wird gemeinsam mit den Datentypen für Parameter, Witness und Ausgabewert sowie dem dazugehörigen CBOR-Protokoll durch die eindeutige Algorithmus-ID gekennzeichnet.

Zu beachten ist dabei, dass in den Request- beziehungsweise Response-Structs die Daten über einen `void*`-Pointer festgelegt sind, also in jedem Fall manuell alloziert und wieder freigegeben werden müssen; auch das Casting sollte von Hilfsfunktionen mit der Namenskonvention `get_myalgorithm_[params|value|witness]([req|res])` übernommen werden. Empfohlen ist bei komplexen Datentypen die Nutzung von Daten-Structs bzw. generell Datentypen mit der Namenskonvention `Myalgorithm[Params|Value|Witness]`.

Damit lässt sich ein Skelett eines Algorithmus wie beispielhaft in Quellcode 5.2 ersichtlich definieren.

```
1  #ifndef MYALGORITHM_H
2  #define MYALGORITHM_H
3  #include <ceral.h>
4
5  CeralAlgorithm myalgorithm;
6
7  typedef int MyalgorithmParams;
8  CeralRequest *new_myalgorithm_request(MyalgorithmParams params);
9  MyalgorithmParams *get_myalgorithm_params(CeralRequest req);
10
11 typedef int MyalgorithmValue;
```

```
12 MyalgorithmValue *get_myalgorithm_value(CeralResponse res);
13
14 typedef int MyalgorithmWitness;
15 MyalgorithmWitness *get_myalgorithm_witness(CeralResponse res);
16
17 #endif // MYALGORITHM_H
```

Quellcode 5.2: myalgorithm.h: Header-Datei für das Skelett eines Beispielalgorithmus

```
1  ////////////////////////////
2  // Helpers & data types //
3  ////////////////////////////
4
5  CeralRequest *new_myalgorithm_request(MyalgorithmParams params) {
6      MyalgorithmParams *params_ptr = malloc(sizeof(MyalgorithmParams));
7      *params_ptr = params;
8      return ceral_new_request(&myalgorithm, params_ptr);
9  }
10 // You can also add a (non-public) new_myalgorithm_witness and/or
       new_myalgorithm_value for use in your executor, or you allocate the
       memory directly in the executor.
11
12 // Getters for types that have to be casted
13 MyalgorithmValue *get_myalgorithm_value(CeralResponse res) {
14     return (MyalgorithmValue*) res->value;
15 }
16 // You should add getters for witness & params just like for value
17
18 ////////////////////////////////
19 // Encoding/Decoding for CBOR //
20 ////////////////////////////////
21
22 CborError myalgorithm_encode_value(CborEncoder *encoder, void *value) {
23     CborError err = CborNoError, err_test;
24
25     // Encode value as CBOR int
26     MyalgorithmValue *unwrapped_value = (MyalgorithmValue*) value;
27     CERAL_HANDLE_CBOR_ERROR(cbor_encode_int(encoder, *value));
28
29     error:
30     return err;
31 }
32
33 CborError myalgorithm_decode_value(CborValue *encoded_value, void **value) {
34     CborError err = CborNoError, err_test;
35
36     // Allocate memory for the value & read from CBOR
37     *value = malloc(sizeof(MyalgorithmValue));
38     CERAL_HANDLE_CBOR_ERROR(cbor_value_get_int(encoded_value, *(
           MyalgorithmValue **)value));
39
40     error:
41     *value = NULL;
42     return err;
43 }
44
45 // You must add decode/encode for witness & params just like for the value
```

```
    here
46
47  ////////////////////////////////
48  // Free all allocated resources //
49  ////////////////////////////////
50
51  void myalgorithm_close_request(CeralRequest *req) {
52      free(res->params);
53  }
54  void myalgorithm_close_response(CeralResponse *res) {
55      free(res->value);
56      free(res->witness);
57  }
58
59  ////////////////////////////////////////////////////////////
60  // Definition of the whole algorithm for use with the framework //
61  ////////////////////////////////////////////////////////////
62
63  // When using the conventional naming schema like here, you can abbreviate
        this block like this:
64  // CeralAlgorithm myalgorithm = { CERAL_ALGORITHM(myalgorithm, "\xD8\x70\x45\
        x83\xc2\x75\x1f\x01") };
65  CeralAlgorithm myalgorithm = {
66          .algorithm_id = (uint8_t*) "\xd8\x70\x45\x83\xc2\x75\x1f\x01",
67          .skeleton = {
68                  .cbor_encode_params = myalgorithm_encode_params,
69                  .cbor_decode_params = myalgorithm_decode_params,
70                  .cbor_encode_value = myalgorithm_encode_value,
71                  .cbor_decode_value = myalgorithm_decode_value,
72                  .cbor_encode_witness = myalgorithm_encode_witness,
73                  .cbor_decode_witness = myalgorithm_decode_witness,
74
75                  .close_request = myalgorithm_close_request,
76                  .close_response = myalgorithm_close_response,
77          },
78          .executor = myalgorithm_execute,
79          .checker = myalgorithm_check,
80  };
```

Quellcode 5.3: myalgorithm.c: Skelett eines Beispielsalgorithmus

Die Methoden für die Serialisierung nutzen die TinyCBOR-Bibliothek[1]. Mit dieser müssen die Serialisierungsmethoden die vom Algorithmus verwendeten Werte für Parameter, Ergebniswert und Witness enkodieren bzw. dekodieren können, sowie beim dekodieren den notwendigen Speicher allozieren. Von *libceral* zusätzlich zur Verfügung gestellt wird das Makro `CERAL_HANDLE_CBOR_ERROR`, welches es deutlich vereinfacht, komplexe Datentypen zu (de)serialisieren: tritt ein Fehler auf, der nicht im Nachhinein behandelt werden kann, wird `err` gesetzt und zur Position `error` gesprungen, wo dieser dann behandelt werden kann. `CborErrorOutOfMemory` ist aktuell der einzige im Nachhinein behandelbare Fehler: tritt dieser auf, zählt TinyCBOR nur noch die notwendigen Bytes, damit diese am Ende realloziert werden können; die Serialisierungsfunktionen werden dann einfach erneut aufgerufen.

[1] `https://github.com/intel/tinycbor/`, abgerufen 11. April 2022

5.3.3 Implementierung von Algorithmen (Executor)

Der Algorithmus selbst besteht aus eine Executor-Funktion, und kann nun das Skelett nutzen, um das Ergebnis sowie das Witness zu errechnen:

```
1  CeralError myalgorithm_execute(CeralRequest *req, CeralResponse *res) {
2  #ifndef MYALGORITHM_NOEXEC
3      MyalgorithmParams *params = get_myalgorithm_params(req);
4      MyalgorithmValue *value = malloc(sizeof(MyalgorithmValue));
5      MyalgorithmWitness *witness = malloc(sizeof(MyalgorithmWitness));
6      // ...
7      res->value = value;
8      res->witness = witness;
9      return CeralNoError;
10 #else
11     return CeralMissingAlgorithmError;
12 #endif
13 }
```

Quellcode 5.4: Beispiel für eine Executor-Funktion

Auch hierbei muss wieder darauf geachtet werden, dass alle allozierten Ressourcen in `close_response` wieder freigegeben werden.

5.3.4 Implementierung von Checkern

Der Checker ist der einfachste Bestandteil eines zertifizierenden Algorithmus - er liest Parameter, Ausgabewert und Witness aus Request und Response, und gibt einfach einen booleschen Wert zurück, der aussagt, ob das Ergebnis korrekt ist.

```
1  bool myalgorithm_checker(CeralRequest *req, CeralResponse *res) {
2      MyalgorithmParams *params = get_myalgorithm_params(req);
3      MyalgorithmValue *value = get_myalgorithm_value(res);
4      MyalgorithmWitness *witness = get_myalgorithm_witness(res);
5      // ...
6      return value == witness;
7  }
```

Quellcode 5.5: Beispiel für eine Checker-Funktion

5.3.5 Formale Verifikation der Checker

Um einen mit *libceral* kompatiblen Checker formal zu verifizieren, können grundsätzlich beliebige C-kompatible Tools genutzt werden, da dieser grundsätzlich eine ganz normale C-Funktion ist. Mit einigen Tools kann es notwendig sein, eine Hilfsfunktion als zugrundeliegende Checker-Methode zu nutzen, damit klassische C-Datentypen ohne `void*`-Pointer genutzt werden können.

Als vermutlich am einfachsten integrierbare Lösung kommt ACSL infrage - hier werden alle für die Beweisführung notwendigen Informationen direkt als Kommentare im Code festgelegt, und dann mit einem Prover automatisch bewiesen [25].

5.4 Eine Registry für konsistente Datentypen

Grundsätzlich ist die genutzte OID jedes Algorithmus dafür sinnvoll, dass unterschiedliche Algorithmen nicht untereinander verwechselt werden können. Denkt man das einen Schritt weiter, können die OIDs dafür genutzt werden, kompatible Algorithmen zu kennzeichnen: der Datentyp ist komplett unabhängig von Checker und Implementierung.

Aus diesem Grund wurde im Rahmen dieser Arbeit die offene „Registry of Certifying Algorithms“[2] ins Leben gerufen, um OIDs, Spezifikationen, Datenstrukturen und Tests für zertifizierende Algorithmen an einem Ort zusammenzufassen. Jeder neue Algorithmus kann hier über einen Pull Request eine eigene OID erhalten; von diesem Algorithmus und dessen Checker können nun verschiedene Versionen auch in verschiedenen Programmiersprachen und für verschiedene Toolchains existieren, die alle aufgrund der identischen Datenstruktur untereinander kompatibel sind.

Dabei ist lediglich zu beachten, dass sowohl Eingabeparameter und Rückgabewert, als auch das Witness Teil dieser Spezifikation sind. Es ist also nicht möglich beispielsweise verschiedene Shortest-Path-Algorithmen zusammenzufassen, wenn Format oder Semantik des Witness nicht übereinstimmen.

Zur Veröffentlichung dieser Arbeit besteht die Registry nur aus einer Struktur und wenigen Beispielen. Aus diesem Grund sollten zertifizierende Algorithmen, die auf das entwickelte Framework oder dessen Serialisierungsprotokoll aufbauen oder dafür adaptiert wurden, hier immer auch mit aufgenommen werden - Ziel ist es, langfristig eine gute Übersicht für Nutzer zu schaffen, welche zertifizierende Algorithmen existieren, was sie tun, und welche Implementierungen davon verfügbar sind.

5.5 Vergleich von zertifizierenden Algorithmen mit Alternativen

Je nach Einsatzzweck von zertifizierenden Algorithmen gibt es unterschiedliche Alternativen mit unterschiedlichen Vor- und Nachteilen, die auch auf unterschiedliche Weise miteinander verbunden werden können.

Für die allgemeine Qualitätssicherung von Software sind der größte konkurrierende Ansatz sicherlich Unit-Tests, beziehungsweise generell klassisches Software-Testing. Dies hat den großen Vorteil, dass es sich nicht nur auf der Ebene von Algorithmen anwenden lässt, sondern über den kompletten Lebenszyklus einer Software - als Test-Driven-Development schon zur Festlegung von Anforderungen, bis hin zu Integration-Tests zur Feststellung der erfolgreichen Markteinführung und im Monitoring sogar darüber hinaus.

Der Nachteil dabei ist, dass nicht geprüfte Fälle eventuell Fehler enthalten können, weshalb die Testfälle einerseits geschickt gewählt sein müssen, und andererseits nie garantiert werden kann, dass ein Fehler nicht im Betrieb doch auftritt. Je nach Sorgfalt funktioniert klassisches Software-Testing überall von unkritischen Systemen bis hin zu lebenskritischen Systemen.

Eine Option beim klassischen Software-Testing, die von immer mehr Testing-Frameworks unterstützt wird, sind Fuzzing Tests. Diese generieren sehr viele zufällige Werte und überprüfen danach, ob das Ergebnis korrekt ist. Zertifizierende Algorithmen lassen sich dafür

[2] `https://codeberg.org/ovgu/ceral-registry`

sehr gut einsetzen, da damit sehr effizient sehr viele Werte getestet werden können.

Sowohl reine zertifizierende Algorithmen als auch bei Projekten mit klassischem Software-Testing ist es meist notwendig, dass Fehler im Nachhinein behoben werden können - ein Satellit beispielsweise, der aufgrund eines Bugs in einem zertifizierenden Algorithmus Montags keine Daten sendet, ist vermutlich nur tolerierbar, wenn ein Software-Update „over-the-air“ möglich ist, um das Problem zu beheben.

Für lebenskritische oder sehr langfristig angelegte Systeme ist die bekannteste Alternative die formale Verifikation von Software. Diese ist deutlich komplexer umzusetzen, da jede einzelne Codezeile auch theoretisch bewiesen werden muss. Auch eine exakte logische Spezifikation des Algorithmus muss ausgearbeitet werden, hierbei können ebenfalls sehr leicht Fehler auftreten, wenn nicht alle Umstände berücksichtigt werden.

Allerdings ist es nicht immer möglich, einen zertifizierenden Algorithmus zu nutzen, da dieser immer erfordert, dass im Fehlerfall auf eine alternative Lösung ausgewichen werden kann - ein Eingreifen durch einen Menschen etwa, oder das Verwerfen der Daten - ohne dass dabei eine Gefahr entsteht. Wenn dies jedoch möglich ist, ist die in Unterabschnitt 5.3.5 geschilderte formale Verifikation von Checkern ein Kompromiss, mit dem garantiert werden kann, dass ein Ergebnis entweder korrekt ist, oder bekannt ist, dass es sich um einen Fehlerfall handelt.

Ein Testaufbau bestehend aus einem Fuzzing-Test eines zertifizierenden Algorithmus mit einem formal verifizierten Checker wäre so die ultimative Lösung, wenn Fehlerfälle zwar theoretisch auftreten dürfen und behandelt oder ignoriert werden können, aber so gut wie möglich vermieden werden sollen.

KAPITEL 6

Fazit

6.1 Zusammenfassung

Es gibt im Bereich von eingebetteten Systemen auf jeden Fall Anwendungsfälle, in denen zertifizierende Algorithmen viel Sinn ergeben können. Darunter fällt, um Punkt FF1 zu beantworten, jegliche Nutzung von Algorithmen, bei der einerseits Fehler eine große Auswirkung haben, und andererseits eine Ausweichmöglichkeit besteht, wenn ein Fehler erkannt wurde. Das Ergebnis kann dann je nach Anwendung entweder verworfen werden, ein zweites Mal berechnet werden (ggf. mit einem anderen Algorithmus), oder es kann darauf durch einen Menschen oder anderweitige Kontrollmechanismen reagiert werden.

Als Alternative kommt vor allem formale Verifikation zum Einsatz, welche zwar theoretisch diese Fehlerbehandlung zur Laufzeit überflüssig macht, aber besonders bei komplexeren Algorithmen deutlich aufwendiger umzusetzen ist - beide Ansätze können kombiniert werden, indem nur der Checker formal verifiziert wird. Eine andere Alternative sind Unit-Tests, die deutlich einfacher zu schreiben sind, bei denen jedoch alle potenziellen Fehlerfälle vorher bekannt sein müssen - die meisten davon vorab zu testen, ist jedoch auch bei zertifizierenden Algorithmen sinnvoll. Fuzzing Unit-Tests können dann direkt vom Checker Gebrauch machen, um auch seltenere Fehlerfälle abzufangen.

Die Output Knowledge sind zunächst die Best Practices aus Abschnitt 5.2 für die Implementierung zertifizierender Algorithmen, welche die Entwicklung dieser generell erleichtern und so in ihrer Gesamtheit Punkt FF2 beantworten. Sie sind so einerseits der Ausgangspunkt für die Framework-Entwicklung, andererseits vor allem für die davon unabhängige Implementierung spezialisierter Algorithmen nützlich. Neben der Interoperatbilität für die Nutzung in mehreren Projekten ist dabei vor allem die Implementierung der Checker relevant - von diesen ist am Ende abhängig, ob es falsch-negative Fehlerfälle gibt, die nicht als solche erkannt werden.

Die entwickelte Solution stellt schließlich das Software-Framework *libceral* aus Kapitel 4 dar - es dient dazu, einen Standard für die Entwicklung sequenzieller zertifizierende Algorithmen zu schaffen, die sowohl auf eingeschränkten eingebetteten Systemen als auch auf performanten Servern in vielfältigen Projekten genutzt werden können, und deren Anfragen und Ergebnisse zwischen unterschiedlichen Systemen über ein Netzwerk übertragen werden kön-

nen. Der in Punkt FF3 vermutete Mehraufwand ist dabei bei der Entwicklung und Nutzung der Algorithmen relativ gering, wie auch in Unterabschnitt 4.2.3 und Unterabschnitt 4.3.3 zu sehen ist - mit dem Framework wurde also ein guter Mittelweg aus guter Nutzerfreundlichkeit auf der einen Seite und geringem Overhead bzw. hoher Interoperabilität auch für eingebettete Systeme auf der anderen gefunden.

Wenn man sich die reinen gemessenen Benchmark-Werte anschaut, stellt sich eventuell ein wenig die Frage, ob ein solches Framework überhaupt notwendig ist, gerade wenn die Serialisierung nicht benötigt wird - es erzeugt schließlich theoretisch unnötigen Overhead. Dies gilt jedoch nur, wenn man ohnehin zertifizierende Algorithmen einsetzen möchte: kennt man seine Anforderungen dabei genau, können die Best Practices helfen, eine darauf spezialisierte Implementierung umzusetzen. Sucht man jedoch allgemein nach Möglichkeiten zur Qualitätssicherung von Algorithmen, bietet ein solches Framework, gegebenenfalls mit Beispielen für ähnliche Algorithmen, einen guten Einstiegspunkt, um mit wenig Aufwand in die Welt der zertifizierenden Algorithmen einzusteigen.

Im Zusammenspiel mit der in Abschnitt 5.4 vorgestellten Registry wird darum so durch die Schaffung eines einheitlichen Standards sowohl die Nutzung als auch die Entdeckbarkeit zertifizierender Algorithmen verbessert.

6.2 Future Work

Die Weiterentwicklung des Frameworks selbst kann vermutlich vor allem aus der Perspektive des Software Engineering betrachtet werden - hier sind stetige Verbesserungen definitiv möglich und wünschenswert, idealerweise lässt sich das Projekt wie auch die dazugehörige Registry als Open-Source-Projekt gemeinschaftlich weiterführen.

Ein mögliches Thema für die weitere Forschung wäre die Evaluation der tatsächlichen Einsetzbarkeit von *libceral* in Projekten, mit besonderem Fokus auf Usability. Dies könnte anhand einer Nutzerstudie erfolgen, idealerweise auch in Projekten, in denen bereits jetzt zertifizierende Algorithmen eingesetzt werden.

Ebenfalls ein relevantes Thema wäre die Anwendbarkeit beziehungsweise Erweiterung des Frameworks auf verteilte zertifizierende Algorithmen, wie bereits in Unterabschnitt 4.3.3 angeschnitten, insbesondere auf eingebetteten Systemen.

Auch die Berücksichtigung zertifizierender Algorithmen in einem allgemeineren methodischen Framework zur Qualitätssicherung bei Algorithmen wäre hierbei spannend - so könnte auch eine Methodik zur Analyse der Eignung bestimmter Ansätze für bestimmte Algorithmen beziehungsweise Einsatzzwecke entwickelt werden.

Grundsätzlich ist das Potenzial zertifizierender Algorithmen als neues Paradigma zur Fehlerbehandlung groß, und das Framework sowie die Best Practices könnten dafür den nächsten großen Schritt hin zu einer Nutzung in Projekten aller Art schaffen.

Literaturverzeichnis

[1] Liu Sida. Google Scholar Trends. https://github.com/liusida/Google-Scholar-Trends, December 2021. (abgerufen am 11. April 2022).

[2] R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, May 2011.

[3] Kim Völlinger. Zertifizierende verteilte Algorithmen. October 2020.

[4] Alan Hevner, Alan R, Salvatore March, Salvatore T, Park, Jinsoo Park, Ram, and Sudha. Design Science in Information Systems Research. *Management Information Systems Quarterly*, 28:75, March 2004.

[5] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, December 2014.

[6] Jan vom Brocke and Alexander Maedche. The DSR grid: Six core dimensions for effectively planning and communicating design science research projects. *Electronic Markets*, 29(3):379–385, September 2019.

[7] Stefan Näher and Kurt Mehlhorn. LEDA: A library of efficient data types and algorithms. In Michael S. Paterson, editor, *Automata, Languages and Programming*, Lecture Notes in Computer Science, pages 1–5, Berlin, Heidelberg, 1990. Springer.

[8] Eyad Alkassar, Sascha Böhme, Kurt Mehlhorn, and Christine Rizkallah. A Framework for the Verification of Certifying Computations. *Journal of Automated Reasoning*, 52(3):241–273, March 2014.

[9] Lorenz Hübschle-Schneider and Peter Sanders. Communication Efficient Checking of Big Data Operations. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 650–659, May 2018.

[10] Algorithmic Solutions. Version 6.6 The LEDA User Manual. http://www.algorithmic-solutions.info/leda_manual/MANUAL.html, April 2020. (abgerufen am 11. April 2022).

[11] Adrian Neumann. *Implementation of Schmidt's Algorithm for Certifying Triconnectivity Testing*. PhD thesis, Universität des Saarlandes Saarbrücken, 2011.

[12] Michaela Iorga, Larry Feldman, Robert Barton, Michael J. Martin, Nedim S. Goren, and Charif Mahmoudi. Fog Computing Conceptual Model. March 2018.

[13] Ellis F. Hitt. Fault-Tolerant Avionics. In *Digital Avionics Handbook*. CRC Press, third

edition, 2015.

[14] J. L. Lions. ARIANE 5 Failure - Full Report. https://www-users.cse.umn.edu/~arnold/disasters/ariane5rep.html, 1996. (abgerufen am 11. April 2022).

[15] J.-M. Jazequel and B. Meyer. Design by contract: The lessons of Ariane. *Computer*, 30(1):129–130, January 1997.

[16] Christoph Lüth, Nicole Megow, Rolf Drechsler, and Udo Frese. Verification for Autonomous Underwater Systems. In Frank Kirchner, Sirko Straube, Daniel Kühn, and Nina Hoyer, editors, *AI Technology for Underwater Robots*, Intelligent Systems, Control and Automation: Science and Engineering, pages 169–181. Springer International Publishing, Cham, 2020.

[17] Ralf Müller. *Selbstüberwachung differenzdruckbasierter Durchflussmessverfahren für Flüssigkeiten*. PhD thesis, September 2005.

[18] Emmanuel Baccelli, Cenk Gundogan, Oliver Hahm, Peter Kietzmann, Martine S. Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C. Schmidt, and Matthias Wahlisch. RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT. *IEEE Internet of Things Journal*, 5(6):4428–4440, December 2018.

[19] C. Bormann and P. Hoffman. Concise Binary Object Representation (CBOR). Technical Report RFC7049, RFC Editor, October 2013.

[20] Linux Kernel Organization. Perf_event_open(2) - Linux manual page. https://www.man7.org/linux/man-pages/man2/perf_event_open.2.html, August 2021. (abgerufen am 11. April 2022).

[21] Valgrind Developers. Massif: A heap profiler. https://valgrind.org/docs/manual/ms-manual.html. (abgerufen am 11. April 2022).

[22] RIOT OS. Oneway malloc. https://doc.riot-os.org/group___oneway___malloc.html, 2022. (abgerufen am 11. April 2022).

[23] ITU-T. X.660 : Information technology - Procedures for the operation of object identifier registration authorities: General procedures and top arcs of the international object identifier tree. https://www.itu.int/rec/T-REC-X.660-201107-I/en, July 2011. (abgerufen am 11. April 2022).

[24] Carsten Bormann. Concise Binary Object Representation (CBOR) Tags for Object Identifiers. Request for Comments RFC 9090, Internet Engineering Task Force, July 2021.

[25] Jens Gerlach. ACSL by Example. November 2020.

Anhang

A.1 Vollständiger Quellcode

Der vollständige Quellcode des Frameworks und sämtlicher Iterationen und Beispiele steht auf zwei Wegen zur Verfügung:

- In der gedruckten Version der Arbeit auf der unten beigelegten CD
- Online unter folgender URL:
  `https://code.ovgu.de/comsys-group/students/msc/2021-marquardt-moritz`

Das finale Framework, zusammen mit allen potenziellen weiteren Fortschritten nach Veröffentlichung dieser Arbeit, ist öffentlich und unter der freien Unlicense-Lizenz hier zu finden:
`https://codeberg.org/ovgu/libceral`

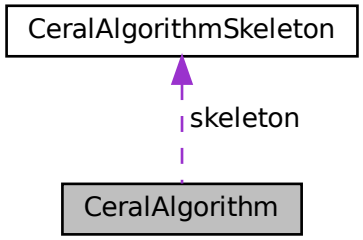
Chapter A.2

Original libceral Documentation

A.2.1 CeralAlgorithm Struct Reference

`#include <ceral.h>`

Collaboration diagram for CeralAlgorithm:



Public Attributes

- uint8_t * algorithm_id
- struct CeralAlgorithmSkeleton skeleton
- bool(* checker)(CeralRequest *req, CeralResponse *res)
- CeralError(* executor)(CeralRequest *req, CeralResponse *res)

A.2.1.1 Detailed Description

A certifying algorithm with all functions required to build, execute and validate a request.

A.2.1.2 Member Data Documentation

A.2.1.2.1 algorithm_id

`uint8_t* CeralAlgorithm::algorithm_id`

An CBOR-encoded ASN.1 OID (RFC 9090) that uniquely identifies the algorithm. You can use https://momar.codeberg.page/oid-tool/?format=cbor-esc&oid=1.3.6.1.4.1.57717.31.0 to format an OID that way, and https://codeberg.org/ovgu/ceral-registry to request an OID for your algorithm.

A.2.1.2.2 checker

`bool(* CeralAlgorithm::checker) (CeralRequest *req, CeralResponse *res)`

A checker implementation that returns true if (and only if) the response is valid for the given request.

Parameters

req	The request that was used to execute the algorithm.
res	The result returned by ceral_execute().

Returns

true if (and only if) the response is valid for the given request.

A.2.1.2.3 executor

`CeralError(* CeralAlgorithm::executor) (CeralRequest *req, CeralResponse *res)`

An executor implementation that runs the algorithm against the request and returns the value and a witness.

Parameters

req	The request that should be evaluated.
res	The response where value and witness should be written to - you may allocate memory here, but make sure that it can be cleaned up with skeleton.close_response()!

Returns

An error if one occured, otherwise CeralNoError.

A.2.1.2.4 skeleton

`struct CeralAlgorithmSkeleton CeralAlgorithm::skeleton`

The skeleton of the algorithm, holding memory management and CBOR (de-)serialization functions.

The documentation for this struct was generated from the following file:

- ceral.h

A.2.2 CeralAlgorithmSkeleton Struct Reference

`#include <ceral.h>`

Public Attributes

- CborError(* cbor_encode_params)(CborEncoder *encoder, void *params)
- CborError(* cbor_decode_params)(CborValue *encoded_value, void **params)
- CborError(* cbor_encode_value)(CborEncoder *encoder, void *value)
- CborError(* cbor_decode_value)(CborValue *encoded_value, void **value)
- CborError(* cbor_encode_witness)(CborEncoder *encoder, void *witness)
- CborError(* cbor_decode_witness)(CborValue *encoded_value, void **witness)
- void(* close_request)(CeralRequest *req)
- void(* close_response)(CeralResponse *res)

A.2.2.1 Member Data Documentation

A.2.2.1.1 cbor_decode_params

`CborError(* CeralAlgorithmSkeleton::cbor_decode_params) (CborValue *encoded_value, void **params)`

Decode a request's parameters from CBOR.

Parameters

encoded_value	The value that holds the CBOR data.
params	The target void** pointer that should be allocated and set to the parsed contents of encoded_value.

Returns

The error that occured, or CborNoError if everything went fine.

A.2.2.1.2 cbor_decode_value

`CborError(* CeralAlgorithmSkeleton::cbor_decode_value) (CborValue *encoded_value, void **value)`

Decode a response value from CBOR.

Parameters

encoded_value	The value that holds the CBOR data.
value	The target void** pointer that should be allocated and set to the parsed contents of encoded_value.

Returns

The error that occured, or CborNoError if everything went fine.

A.2.2.1.3 cbor_decode_witness

`CborError(* CeralAlgorithmSkeleton::cbor_decode_witness) (CborValue *encoded_value, void **witness)`

Decode a response witness from CBOR.

Parameters

encoded_value	The value that holds the CBOR data.
witness	The target void** pointer that should be allocated and set to the parsed contents of encoded_value.

Returns

The error that occured, or CborNoError if everything went fine.

A.2.2.1.4 cbor_encode_params

`CborError(* CeralAlgorithmSkeleton::cbor_encode_params) (CborEncoder *encoder, void *params)`

Encode a requests's parameters to CBOR.

Parameters

encoder	The encoder that should be used.
params	The void* pointer to the field that should be encoded.

Returns

The error that occured, or CborNoError if everything went fine.

A.2.2.1.5 cbor_encode_value

`CborError(* CeralAlgorithmSkeleton::cbor_encode_value) (CborEncoder *encoder, void *value)`

Encode a response value to CBOR.

Parameters

encoder	The encoder that should be used.
value	The void* pointer to the field that should be encoded.

Returns

The error that occured, or CborNoError if everything went fine.

A.2.2.1.6 cbor_encode_witness

```
CborError(* CeralAlgorithmSkeleton::cbor_encode_witness) (CborEncoder *encoder, void
*witness)
```

Encode a response witness to CBOR.

Parameters

encoder	The encoder that should be used.
witness	The void* pointer to the field that should be encoded.

Returns

The error that occured, or CborNoError if everything went fine.

A.2.2.1.7 close_request

```
void(* CeralAlgorithmSkeleton::close_request) (CeralRequest *req)
```

Free all memory that is allocated by either new_myalgorithm_request() or cbor_decode_params().

Parameters

req	The request object that holds the parameters.

A.2.2.1.8 close_response

```
void(* CeralAlgorithmSkeleton::close_response) (CeralResponse *res)
```

Free all memory that is allocated by either executor() or cbor_decode_value() and cbor_decode_witness().

Parameters

res	The response object that holds the value and the witness.

The documentation for this struct was generated from the following file:

- ceral.h

A.2.3 CeralRequest Struct Reference

`#include <ceral.h>`

Public Attributes

- uint8_t * algorithm_id
- uint64_t request_id
- void * params
- CeralError error

A.2.3.1 Detailed Description

A request designated for a certifying algorithm.

A.2.3.2 Member Data Documentation

A.2.3.2.1 algorithm_id

`uint8_t* CeralRequest::algorithm_id`

The name of the algorithm, optionally with a version.

A.2.3.2.2 error

`CeralError CeralRequest::error`

After parsing a request or response, this contains the CBOR error and MUST be checked against CborNoError.

A.2.3.2.3 params

`void* CeralRequest::params`

The parameters passed to the executor.

A.2.3.2.4 request_id

`uint64_t CeralRequest::request_id`

A random (or incremental when used in a thread-safe manner on a single node) request ID.

The documentation for this struct was generated from the following file:

- ceral.h

A.2.4 CeralResponse Struct Reference

`#include <ceral.h>`

Public Attributes

- uint8_t * algorithm_id
- uint64_t request_id
- void * value
- void * witness
- CeralError error

A.2.4.1 Detailed Description

A response from a certifying algorithm, containing the calculated value, as well as the request data.

A.2.4.2 Member Data Documentation

A.2.4.2.1 algorithm_id

```
uint8_t* CeralResponse::algorithm_id
```

A CBOR-encoded OID that uniquely identifies the algorithm. You can use https://momar.codeberg.↩page/oid-tool/?format=cbor-esc&oid=1.3.6.1.4.1.57717.31.1 to build this id.

A.2.4.2.2 error

```
CeralError CeralResponse::error
```

After parsing a request or response, this contains the CBOR error and MUST be checked against CborNoError.

A.2.4.2.3 request_id

```
uint64_t CeralResponse::request_id
```

A random (or incremental when used in a thread-safe manner on a single node) request ID.

A.2.4.2.4 value

```
void* CeralResponse::value
```

The return value calculated by the certifying algorithm.

A.2.4.2.5 witness

```
void* CeralResponse::witness
```

The matching witness calculated by the certifying algorithm.

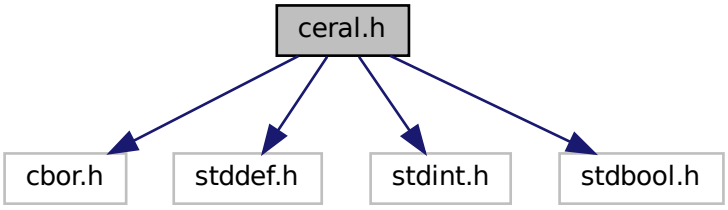
The documentation for this struct was generated from the following file:

- ceral.h

A.2.5 ceral.h File Reference

```
#include <cbor.h>
#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>
```

Include dependency graph for ceral.h:



Classes

- struct CeralRequest
- struct CeralResponse
- struct CeralAlgorithmSkeleton
- struct CeralAlgorithm

Macros

- #define CERAL_GENERATE_REQUEST_ID rand()
- #define CERAL_HANDLE_CBOR_ERROR(x)
- #define CERAL_ALGORITHM(name, oid)
- #define CERAL_CBOR_DEFAULT_BUFFER_SIZE 128 ∗ sizeof(uint8_t)

Typedefs

- typedef enum CeralError CeralError
- typedef struct CeralRequest CeralRequest
- typedef struct CeralResponse CeralResponse
- typedef struct CeralAlgorithmSkeleton CeralAlgorithmSkeleton
- typedef struct CeralAlgorithm CeralAlgorithm

Enumerations

- enum CeralError { CeralNoError = CborNoError, CeralIllegalInputError = 31400, CeralMissingAlgorithmError = 31404, CeralAlgorithmInternalError = 31500 }

Functions

- size_t ceral_encode_request (CeralAlgorithm *alg, CeralRequest *req, uint8_t *buffer, size_t buffer_size)
- size_t ceral_encode_response (CeralAlgorithm *alg, CeralResponse *res, uint8_t *buffer, size_t buffer_size)
- CeralRequest * new_ceral_request_from_cbor (CeralAlgorithm *alg, uint8_t *buffer, size_t size)
- CeralResponse * new_ceral_response_from_cbor (CeralAlgorithm *alg, uint8_t *buffer, size_t size)
- void ceral_execute (CeralAlgorithm *alg, CeralRequest *req, CeralResponse *res)
- bool ceral_check (CeralAlgorithm *alg, CeralRequest *req, CeralResponse *res)
- CeralRequest * new_ceral_request (CeralAlgorithm *algorithm, void *params)
- CeralResponse * new_ceral_response (CeralAlgorithm *alg, CeralRequest *req)
- void close_ceral_response (CeralAlgorithm *alg, CeralResponse *res)
- void close_ceral_request (CeralAlgorithm *alg, CeralRequest *req)
- bool ceral_algorithm_id_equals (uint8_t *id_a, uint8_t *id_b)

A.2.5.1 Macro Definition Documentation

A.2.5.1.1 CERAL_ALGORITHM

```
#define CERAL_ALGORITHM(
            name,
            oid )
```

Value:

```
    .algorithm_id = (uint8_t*) oid,\
    \
    .skeleton = {\
        .cbor_encode_params = name ## _encode_params,\
        .cbor_decode_params = name ## _decode_params,\
        .cbor_encode_value = name ## _encode_value,\
        .cbor_decode_value = name ## _decode_value,\
        .cbor_encode_witness = name ## _encode_witness,\
        .cbor_decode_witness = name ## _decode_witness,\
        \
        .close_request = close_ ## name ## _request,\
        .close_response = close_ ## name ## _response,\
    },\
    .executor = name ## _executor,\
    .checker = name ## _checker,
```

CERAL_ALGORITHM(name, oid) fills the fields of a CeralAlgorithm according to the usual naming convention. It requires a name (that's used for the method names) and an algorithm id.

Usual usage would look like this: CeralAlgorithm myalgorithm = { CERAL_ALGORITHM(myalgorithm, "\x↩
D8\x70\x45\x83\xc2\x75\x1f\x01") };

A.2.5.1.2 CERAL_CBOR_DEFAULT_BUFFER_SIZE

```
#define CERAL_CBOR_DEFAULT_BUFFER_SIZE 128 * sizeof(uint8_t)
```

A sensible default buffer size for CBOR-encoded data.

A.2.5.1.3 CERAL_GENERATE_REQUEST_ID

```
#define CERAL_GENERATE_REQUEST_ID rand()
```

CERAL_GENERATE_REQUEST_ID shall return a random uint64 number to identify a request. This is by default done by using the `rand()` function, but ideally should identify the system somehow together with an incremental request ID.

A.2.5.1.4 CERAL_HANDLE_CBOR_ERROR

```
#define CERAL_HANDLE_CBOR_ERROR(
            x )
```

Value:

```
    err_test = x; \
    if (err_test != CborNoError) { \
        err = err_test; \
        if (err != CborErrorOutOfMemory) goto error; \
    }
```

CERAL_HANDLE_CBOR_ERROR(CborError x) uses the variables `err` and `err_test`, and uses `goto error` to jump to an error handler if there's a non-recoverable error. A usual use case could look like this:

CborError err = CborNoError, err_test; CERAL_HANDLE_CBOR_ERROR(...); error: return err;

A.2.5.2 Typedef Documentation

A.2.5.2.1 CeralAlgorithm

```
typedef struct CeralAlgorithm CeralAlgorithm
```

A certifying algorithm with all functions required to build, execute and validate a request.

A.2.5.2.2 CeralAlgorithmSkeleton

```
typedef struct CeralAlgorithmSkeleton CeralAlgorithmSkeleton
```

A.2.5.2.3 CeralError

```
typedef enum CeralError CeralError
```

An error that could occur during an execution or deserialization. It might also be a CborError if the values are different.

A.2.5.2.4 CeralRequest

```
typedef struct CeralRequest CeralRequest
```

A request designated for a certifying algorithm.

A.2.5.2.5 CeralResponse

```
typedef struct CeralResponse CeralResponse
```

A response from a certifying algorithm, containing the calculated value, as well as the request data.

A.2.5.3 Enumeration Type Documentation

A.2.5.3.1 CeralError

```
enum CeralError
```

An error that could occur during an execution or deserialization. It might also be a CborError if the values are different.

Enumerator

CeralNoError	The action succeeded without issues.
CeralIllegalInputError	The preconditions weren't matched.
CeralMissingAlgorithmError	The algorithm is missing, either because the OIDs don't match or because the executor wasn't compiled.
CeralAlgorithmInternalError	An unknown algorithm-internal error occured.

A.2.5.4 Function Documentation

A.2.5.4.1 ceral_algorithm_id_equals()

```
bool ceral_algorithm_id_equals (
            uint8_t * id_a,
            uint8_t * id_b )
```

Check if two algorithm IDs are equal.

Parameters

id_a	The first ID
id_b	The second ID

Returns

true if the IDs are equal

A.2.5.4.2 ceral_check()

```
bool ceral_check (
            CeralAlgorithm * alg,
            CeralRequest * req,
            CeralResponse * res )
```

Validate a response from a certifying algorithm using the matching checker.

Parameters

alg	The algorithm implementation to use.
req	The request that was used to produce the response.
res	The response to check, must include the request with all parameters.

Returns

true if the checker deemed the result valid.

A.2.5.4.3 ceral_encode_request()

```
size_t ceral_encode_request (
            CeralAlgorithm * alg,
            CeralRequest * req,
            uint8_t * buffer,
            size_t buffer_size )
```

Encode a request for transmission over any binary protocol.

Parameters

alg	The algorithm implementation to use.
req	The request to encode.
buffer	The buffer where to write the result.
buffer_size	The size of buffer.

Returns

The length of the encoded request in bytes.

A.2.5.4.4 ceral_encode_response()

```
size_t ceral_encode_response (
            CeralAlgorithm * alg,
            CeralResponse * res,
            uint8_t * buffer,
            size_t buffer_size )
```

Encode a response for transmission over any binary protocol.

Parameters

alg	The algorithm implementation to use.
res	The response to encode.
buffer	The buffer where to write the result.
buffer_size	The size of buffer.

Returns

The length of the encoded request in bytes.

A.2.5.4.5 ceral_execute()

```
void ceral_execute (
            CeralAlgorithm * alg,
            CeralRequest * req,
            CeralResponse * res )
```

Execute a request with the matching executor locally.

Parameters

alg	The algorithm implementation to use.
req	The request to execute.
res	The response where the value and witness should be written to.

A.2.5.4.6 close_ceral_request()

```
void close_ceral_request (
            CeralAlgorithm * alg,
            CeralRequest * req )
```

Free all resources allocated by a request.

Parameters

alg	The algorithm implementation to use.
req	The request of a certifying algorithm.

A.2.5.4.7 close_ceral_response()

```
void close_ceral_response (
            CeralAlgorithm * alg,
            CeralResponse * res )
```

Free all resources allocated by a response and its attached response.

Parameters

alg	The algorithm implementation to use.
res	The response of a certifying algorithm.

A.2.5.4.8 new_ceral_request()

```
CeralRequest* new_ceral_request (
            CeralAlgorithm * algorithm,
            void * params )
```

Allocate a new request on the heap. Calling ceral_close_request() later is required!

Parameters

algorithm	The algorithm implementation to use.
params	The input parameters to use for the algorithm.

A.2.5.4.9 new_ceral_request_from_cbor()

```
CeralRequest* new_ceral_request_from_cbor (
            CeralAlgorithm * alg,
            uint8_t * buffer,
            size_t size )
```

Decode a request or response transmitted via a binary protocol for further processing.

Parameters

alg	The algorithm implementation to use.
buffer	The request or response in its CBOR-encoded binary format.
size	The size of buffer.

Returns

The decoded response, containing the request.

A.2.5.4.10 new_ceral_response()

```
CeralResponse* new_ceral_response (
            CeralAlgorithm * alg,
            CeralRequest * req )
```

Allocate a new response on the heap. Calling ceral_close_response() later is required!

Parameters

alg	The algorithm implementation to use.
req	The request to create a response for.

A.2.5.4.11 new_ceral_response_from_cbor()

```
CeralResponse* new_ceral_response_from_cbor (
            CeralAlgorithm * alg,
            uint8_t * buffer,
            size_t size )
```

Decode a request or response transmitted via a binary protocol for further processing.

Parameters

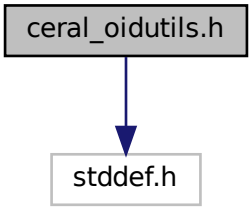
alg	The algorithm implementation to use.
buffer	The request or response in its CBOR-encoded binary format.
size	The size of buffer.

Returns

The decoded response, containing the request.

A.2.6 ceral_oidutils.h File Reference

`#include <stddef.h>`
Include dependency graph for ceral_oidutils.h:



Functions

- size_t ceral_oid_length (uint8_t *oid)
- size_t ceral_oid_prefix_byte_count (uint8_t *oid)
- size_t ceral_algorithm_id_length (uint8_t *id)
- int ceral_algorithm_id_str (char *str, uint8_t *id)

A.2.6.1 Function Documentation

A.2.6.1.1 ceral_algorithm_id_length()

```
size_t ceral_algorithm_id_length (
            uint8_t * id )
```

Get the length of an algorithm ID in bytes, consisting of a full RFC 9090 encoded OID.

Parameters

id	The CBOR-encoded OID according to RFC 9090.

Returns

The length of the ID, effectively the sum of ceral_oid_prefix_byte_count(id) and ceral_oid_length(id).

A.2.6.1.2 ceral_algorithm_id_str()

```
int ceral_algorithm_id_str (
            char * str,
            uint8_t * id )
```

Write the OID to a string (which must have a large enough length) as hexadecimal bytes (so, 2∗ceral_oid_↩ length(id)+1).

Parameters

str	The string to write the OID to.
id	The ID to write into the string.

Returns

The number of copied bytes (and thus normally the length of the stringified OID).

A.2.6.1.3 ceral_oid_length()

```
size_t ceral_oid_length (
            uint8_t * oid )
```

Get the length of a CBOR-encoded OID (without the prefix bytes).

Parameters

oid	The CBOR-encoded OID according to RFC 9090.

Returns

The length of the OID.

A.2.6.1.4 ceral_oid_prefix_byte_count()

```
size_t ceral_oid_prefix_byte_count (
            uint8_t * oid )
```

Get the number of bytes to skip in a CBOR OID from its length.

Parameters

oid	The CBOR-encoded OID according to RFC 9090.

Returns

The length of the OID prefix.

Hiermit versichere ich, dass die vorliegende Arbeit mit dem Titel *Interoperable zertifizierende Algorithmen auf eingebetteten Systemen* selbstständig verfasst worden ist, dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien – dem Wortlaut oder Sinn nach entnommenen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind.

Magdeburg, 10. November 2022

(Moritz Marquardt)