Hochschule Offenburg
offenburg.university

Konzeptionierung, Implementierung und Validierung einer Rust basierten Software-Bibliothek zum Management von Zertifikaten in Feldbusgeräten

Bachelorthesis

für die Prüfung zum

Bachelor of Science

im Studiengang Mechatronik und autonome Systeme

Fakultät Elektrotechnik, Medizintechnik und Informatik

an der Hochschule für Technik, Wirtschaft und Medien Offenburg

von

Marius Urban

27.Februar 2023

Bearbeitungszeitraum	6 Monate
1. Gutachter	Prof. Dr.-Ing. Axel Sikora
2. Gutachter	Prof. Dr. rer. nat. Stefan Wehr

Kurzfassung

In modernen Industrieautomatisierungssysteme kann die IT-Sicherheit nicht mehr ignoriert werden. Um dem Datenverkehr Schutz zu bieten, sind kryptografische Schutzmaßnahmen notwendig. Eine gängige Schutzmaßnahme ist die Verwendung von digitalen Zertifikaten zur Autorisierung und Authentifizierung. Um Zertifikate sicher und geregelt auf Endgeräte zu bringen, ist jedoch eine Public-Key-Infrastructure notwendig. Solche PKIs sind bisher wenig im Umfeld der Industrieautomatisierung untersucht. Das Institut für verlässliche Embedded-Systems der Hochschule Offenburg bietet hierfür eine mögliche Lösung, welche auf einer zentralen Einheit, genannt Credentialing Entity, basiert. Ein Demonstrator dieses Konzepts wurde bereits in den weit verbreiteten Systemprogrammier-sprachen C und C++ implementiert.

Im Rahmen dieser Arbeit wird die Verwendung der modernen speichersicheren Programmiersprache Rust in der Systemprogrammierung als Alternative zu den Domänenführern C/C++ am Beispiel der Implementierung der Credentialing-Entity untersucht. Hierbei werden Aspekte wie die Vorzüge Rusts, dessen Ökosystem und Interoperabilität mit den Marktführern C/C++ untersucht.

Abstract

In modern industrial automation IT security can no longer be ignored. Cryptographic protection of exchanged data is necessary. A common approach is the use of digital certificates, to achieve authorization and authentication. To safely distribute and manage these certificates a public key infrastructure is necessary. Such pkis are currently underexplored in the field of industrial automation. The institute for reliable embedded systems of the Hochschule Offenburg offers on possible solution to this problem by using a central unit called credentialing entity. As a proof of concept a demonstrator of this concept is implemented in the popular systems programming languages C and C++.

In the scope of this thesis the use of the modern memory safe systems programming language rust as an alternative to the domain leader C/C++ in systems programming is explored using the example of the implementation of the credentialing entity. Here the positive aspects of the rust language, its ecosystem and the interoperability with current domain leader C/C++ are examined.

Inhalt

Inhalt 1
1 Einleitung 1
1.1 Field-PKI 1
1.2 Rust 2
1.3 Zielsetzung 2
1.4 Vorgehen 2
1.5 Aufbau 3
2 . Rust 4
2.1 Speichersicherheit 4
2.1.1 Ownership 5
2.1.2 Referenzen: 6
2.1.3 Lifetimes: 7
2.2 Foreign-Function-Interfaces in Rust 8
2.2.1 C-Code aus Rust-Code aufrufen 8
2.2.2 Rust-Code aus C-Code aufrufen 13
2.3 Nützliche Rust-Konzepte 16
2.3.1 Rust-Enum 16
3 Architektur 18
3.1 Aufbau und Struktur 18
3.2 Logik und Ablauf 21
3.2.1 Hauptschleife 22
3.2.2 Grobablauf CE-Komponente 23
3.2.3 Verbindungsaufbau 24
3.2.4 Ablauf Bewirtschaftung 25
4 Bibliotheken 27
4.1 TLS-Bibliotheken 29

4.1.1 Rustls ... 30
4.1.2 Embedded-TLS ... 30
4.1.3 WebRTC-DTLS ... 30
4.1.4 SaiTLS ... 31
4.1.5 Fazit ... 31
4.2 X.509-Zertifikats-Bibliothek ... 31
4.2.1 rpki ... 32
4.2.2 X509-cert ... 32
4.2.3 barebones-x509 ... 32
4.2.4 x509-certificate ... 32
4.2.5 picky ... 33
4.2.6 X.509 Parser ... 33
4.2.7 rcgen ... 33
4.2.8 Fazit ... 34
5 Rust-Architektur ... 35
5.1 Schwierigkeiten und Probleme bei der Portierung ... 35
5.2 Grobarchitektur Gesamtsystem ... 36
5.3 Architektur Rust-CE ... 37
6 Umsetzung ... 39
6.1 Ringpuffer ... 39
6.1.1 Wie funktioniert ein Ringpuffer ... 39
6.1.2 Implementierung ... 40
6.2 CA ... 41
6.2.1 CA-Trait ... 42
6.2.2 Implementierung interne CA ... 43
6.3 CE ... 45
6.3.1 Struktur ... 45

6.3.2 Busschnittstelle ... 47
6.3.3 Bewirtschaftungslogik ... 49
6.3.4 C-Schnittstelle ... 49
6.4 Integration ... 51
7 Validierung ... 53
7.1 Herausforderungen beim Testen ... 53
7.2 Komponententests ... 53
7.2.1 MessageDe/Encoder und internalCa ... 53
7.2.2 CAServices ... 55
7.3 Integrationstests ... 55
8 Zusammenfassung ... 57
9 Abkürzungsverzeichnis ... 59
9.1 Abkürzungen ... 59
9.2 Begriffe und Definitionen ... 59
10 Anhang ... 61
10.1 Untersuchung TLS ... 62
10.2 Untersuchung Zertifikatsbibliotheken ... 63
11 Literatur ... 64

1 Einleitung

Industrieautomatisierungsnetze (IA-Netze) wurden lange Zeit als ein Problem einzig auf Ebene der Operational Technologies (OT) betrachtet, welche sich Einzig um die Regelung, Steuerung und Überwachung in einem geschlossenen System kümmerten. Steigende Anforderungen an IA-Netze stark angetrieben durch Visionen der Industrie 4.0 sorgte dafür, dass reine OT-Netze nun mit IT(Information Technology)-Netzen verbunden werden [1]. Diese Erweiterung eröffnet neben neuen Möglichkeiten auch neue Risiken vor allem auf Ebene der Informationssicherheit. Schutzziele wie Vertraulichkeit, Integrität und Authentizität [2, 3] sind hierbei nicht mehr gegeben. Diese Schwächen sind in der Industrie jedoch bekannt, und Lösungen werden angestrebt. Eine häufiger Lösungsansatz stellt die Verwendung von Zertifikaten dar. Was allerdings bisher außer Acht gelassen worden ist, ist die Art und Weise wie solche Zertifikate auf die Endgeräte auch IA-Komponenten kommt. Wie eine solche sogenannte Public-Key-Infrastructure(PKI) aussehen könnte, ist Gegenstand des FieldPKI-Projekt des Instituts für verlässliche Embedded Systems (ivESK) der Hochschule Offenburg[4, 5].

1.1 Field-PKI

Das Field-PKI-Modell sieht für das Management der Zertifikate eine Einheit vor, die Credentialing-Entity (CE) heißt. Es ist in einer technologieunabhängig definiert. Diese CE hat primär drei Aufgaben. Erstens müssen neue IA-Komponenten überprüft und als Teil des Systems akzeptiert werden. Dieser Schritt wird Inbesitznahme genannt. In einem zweiten Schritt wird eine gemeinsame Vertrauensbasis geschaffen, um der neuen IA-Komponente zu ermöglichen mit anderen Komponenten zu kommunizieren. Hierbei werden notwendige gültige Zertifikate und Vertrauensanker auf die Komponenten aufgebracht. Vertrauensanker sind hierbei Zertifikate, die es ermöglichen die Zertifikate der Endknoten zu validieren. Ist dies erfolgreich geschehen muss, dieser Zustand aufrechterhalten werden und ungültige Zertifikate erneuert werden.

Um dieses Konzept zu demonstrieren, wurde ein Demonstrator erstellt, der den Prozess des Aufbringens von Zertifikaten und Vertrauensankern auf eine IA-Komponente simuliert. Hierfür wurde jeweils eine Anwendung für die CE und für eine IA-Komponente in C/C++ geschrieben. Im Demonstrator wurde sich für den

CAN-Bus mit dem CANopen-Kommunikationsprotokoll entschieden. Zur Authentifizierung und Verschlüsselung der Kommunikation wird DTLS(Datagram Transport Layer Security) verwendet.

1.2 Rust

Rust ist eine aufstrebende Systemprogrammiersprache, welche die Probleme der Speichersicherheit in den traditionellen Systemprogrammiersprachen C und C++ angeht, ohne hierbei auf Performance zu verzichten. In Kombination mit modernen Sprachfunktionen schaffte Rust den Domänenführe anzugreifen und sich prominente Unterstützer in der Industrie sichern wie zum Beispiel AmazonWebServices[6].

1.3 Zielsetzung

Ziel dieser Arbeit ist der ist eine Alternative Implementierung der Credentialing-Entity bereitzustellen, welche in der Programmiersprache Rust geschrieben ist. Diese Implementierung soll als statische Bibliothek bereitgestellt werden, die eine C-Schnittstelle hat. So kann diese Alternativ zur C++-CE in der Applikation verwendet werden.

1.4 Vorgehen

Das Vorgehen bei dieser Arbeit besteht ausfolgenden Schritten:

- Einarbeitung in das Thema Zertifikate und TLS
- Untersuchung der C/C++-Credentialing-Entity. Hier wird sowohl die Programmlogik herausgearbeitet als auch eine Liste an Komponenten und Funktionen erstellt, die für die Rust-Implementierung benötigt werden
- Erstellung einer Liste an Funktionen für die eine Bibliothek verwendet werden soll. Anschließend wird das Rust-Ökosystem auf Bibliotheken mit jenen Funktionen untersucht, Optionen verglichen und sich für die beste Bibliothek entschieden.
- Untersuchung der Interoperabilität von Rust und C
- Erstellung einer Architektur für die Rust-CE

- Implementierung der Rust-Bibliothek und Erstellung der C-Schnittstelle
- Einbau der Rust-Bibliothek in die vorhandene C-Anwendung und Anpassung des Buildscripts
- Test und Validierung der Applikation

1.5 Aufbau

In den folgenden Kapiteln werden diese Themen behandelt:

2. Rust
   Dieses Kapitel bietet eine Einführung in die Programmiersprache Rust. Hierbei wird vor allem auf Rusts Vorteile in der Speichersicherheit und die Interoperabilität mit C/C++ eingegangen.

3. Architektur
   In diesem Kapitel wird die Architektur des vorhandenen C++-Demonstrators analysiert.

4. Bibliotheken
   Dieses Kapitel beschäftigt sich mit Drittbibliotheken. Zunächst werden benötigte Funktionalitäten zusammengetragen. Anschließen werden die Ergebnisse der Recherche in das Rust-Ökosystem präsentiert und ausgewertet.

5. Rust-Architektur
   Dieses Kapitel beschäftigt sich mit der Architektur der Rust-Implementierung.

6. Umsetzung
   Dieses Kapitel beschäftigt sich mit der Implementierung der einzelnen Komponenten der CE, der Integration der Bibliothek in das Gesamtsystem und der Verifikation der Implementierung.

7. Validierung
   Dieses Kapitel beschäftigt sich mit den durchgeführten Test um die Korrektheit der Implementierung zu validieren.

2 . Rust

Dieses Kapitel gibt einen kurzen Einblick in die Rust-Programmiersprache. Hierbei wird zunächst auf Speichersicherheit in Rust eingegangen. Im Anschluss wird die Interoperabilität zwischen C und Rust näher beleuchtet. Abschließend wird auf nützliche Sprachkonzepte von Rust eingegangen, welche während der Implementierung verwendet werden.

Rust ist eine kompilierte Multiparadigmen-Systemprogrammiersprache [7]. Die Sprache legt besonderen Wert auf Sicherheit und Parallelität. Um den Anforderungen der hohen Leistung und der Speichersicherheit gerecht zu werden, setzt Rust hierbei nicht auf einen Garbage-Collector wie viele andere speichersichere Hochsprachen, sondern auf einen spezielles Typensystem. Dieses ermöglicht die Speichersicherheit zur Kompilierungszeit zu verifizieren. Gleichzeitig ermöglicht Rust durch objektorientierte und funktionale Konzepte einen hohen Abstraktionsgrad, die durch Zero-Cost-Abstractions kaum oder keine Leistungs- oder Speichereinbußen zur Folge haben [8].

2.1 Speichersicherheit

Rusts besonderes Typensystem versucht folgende Fehlerklassen und deren Ursachen zu eliminieren:

- Zugriff auf ungültigen Speicher
    - Erzeugung zufälliger Zeiger, Zeigerarithmetik und verwenden/freigeben von zuvor freigegebenem Speicher
- Speicherlecks (Memory-Leaks)
    - Heap-reservierter Speicher wird nicht wieder freigegeben
- Pufferüberläufe (Buffer-Overflows)
    - Speicherlecks und fehlende Prüfung der grenzen
- Data-Races
    - Unkontrollierter Speicherzugriff durch mehrere Akteure

Um dieses Ziel zu erreichen, gibt es eine Reihe von Konzepten, die es ermöglicht diese Ziele zur Kompilierungszeit umzusetzen.

2.1.1 Ownership

Laut Microsofts internen Sicherheitsteam MSRC sind 70% der Sicherheitskritischen Schwachstellen in Microsofts C/C++-Codebasis auf Speichersicherheitsfehler zurückführen [9]. Eine Ursache hierfür kann sein, dass vergessen wird den Speicher freizugeben. Eine weitere Möglichkeit ist, dass mehrere Akteure Zugriff auf den Speicher haben und nicht klar ist wer die Verantwortung für das Freigeben des Speichers hat. Für beide dieser Szenarien bietet Rusts Ownership-System eine Lösung.

Das Ownership-System besteht aus 3 grundlegenden Regeln [10]:

1. Alle Daten haben einen Besitzer
2. Für diese Daten gibt es je exakt einen Besitzer
3. Wird der Gültigkeitsbereich des Besitzers verlassen, wir der Speicher freigegeben

Diese 3 Prinzipien sind in dem untenstehenden Listing 1 dargestellt:

```
1  {
2      let mut i: i32;
3      i = 5;
4  //  ^---^
5  // i is now owner of the value 5
6      let j = takes_val(i);
7  //          ^---------^
8  // function takes ownership from i and gives it to j
9  // i += 1; !!!!
10 // invalid because i no longer owns the integer
11 }
12 // j out of scope and the value is dropped
```

Listing 1: Beispiel Rust-Ownership

Zunächst wird die Variable i in Zeile zwei erzeugt. In der darauffolgenden Zeile wird der Wert 5 erzeugt und i wird zum Besitzer gemacht. In Zeile sechs passieren zwei Dinge: i wird als Werteparameter übergeben. Hierbei gibt i die Besitzrechte über die 5 and die Funktion ab. Die aufgerufene Funktion erzeugt nun einen Wert, gibt diesen an die neue Variable j und macht diese zum Besitzer.

Zeile neun zeigt etwas Unmögliches. i hat den Besitz über den Integer an die Funktion abgegeben und enthält nun nichts mehr. Würde man versuchen dies zu kompilieren, würde der Compiler diesen Bruch der Ownership-Regeln als Fehler zurückmelden.

Die dritte Ownership-Regel wird in Zeile zwölf demonstriert. j besaß noch Daten, während das Programm dessen Gültigkeitsbereich verlässt. Hierbei wird nun der Speicher für diese Variable freigegeben.

2.1.2 Referenzen:

In einer unsicheren Systemprogrammiersprache wie C gibt es keine strengen Regeln, was die Erzeugung und Veränderung von Zeigern angeht. So ist es möglich Null-Zeiger und Zeiger auf zufälligen oder ungültigen Speicher zu erzeugen. Auch ist das Verändern der Speicheradresse, auf die ein Pointer zeigt, möglich, ohne die Gültigkeit der neuen Adresse zu validieren wie in Listing 2 zu sehen.

```
1  //create pointer to invalid memory adress
2  int nullptr = 0x0;
3  /*
4  ......
5  */
6  int a = 5;
7  //create valid pointer
8  int *ptr = &a;
9  //unsound pointer arithmetic
10 ptr++;
11 //undefined behaviour
12 int b = *ptr;
```

Listing 2: gefährliche Zeigerarithmetik im C

Rust bedient sich hier dem Konzept der Referenzen, welches auch in C++ zu finden ist. Referenzen sind auch Pointer allerdings können sie nur auf gültige Variablen erzeugt werden. Auch ist keine Veränderung der Adresse auf die Referenz zeigt möglich. Referenzen helfen somit die Probleme mit ungültigen Zeigern in Listing 2 zu verhindern.

```
1  let a: u32 = 5;
2  let b = &a;
3  //$ Print 5
4  println!("Print {}",b)
5  //creating reference on call
6  takes_a_ref(&a);
7  //passing a reference directly
8  takes_a_ref(b);
9  // Function that takes a reference as a parmeter
10 fn takes_a_ref(reference: &u32) {/* ... */}
```

Listing 3: Referenzen in Rust

Referenzen werden mit Und-Zeichen (&) markiert. Dies gilt sowohl für den Datentyp wie zum Beispiel in einer Funktionssignatur (siehe Listing 3) oder auch das Erzeugen einer Referenz. Soll der Besitzer der Referenz die Daten verändern können muss der Deklaration das Keyword "mut"(mutable) vorangestellt werden. Referenzen besitzen jedoch Einschränkung.

```
1  let mut a = 5;
2  {
3      //multiple immutable references allowed
4      let b = &a;
5      let c = &a;
6  }
7  /* ... */
8  {
9      let b = &mut a;
10     //only one mutable reference allowed
11     //let c = &mut a;
12 }
```

Listing 4: Regeln zu Referenzen

Es kann entweder eine Referenz auf veränderbare Daten oder beliebig viele nicht veränderbare Referenzen geben (siehe Listing 4). Somit kann dem Besitzer jeglicher Referenzen garantiert werden, dass die referenzierten Daten nicht unerwartet verändert werden.

2.1.3 Lifetimes:

Man betrachte die folgende Listing 5.

```
1  //scope A start
2  let r;
3
4  {
5       //scope B starts
6       let x = 5;
7       r = &x;
8       //scope B ends
9  }
10 println!("r: {}", r);
11 //scopeA ends
```

Listing 5: ungültige Lifetime[11]

Nachdem die Variable r deklariert wird, wird in einem Block eine weitere Variable x mit dem Wert 5 instanziiert. Eine Referenz auf x wird r zugewiesen und der Block und damit der Gültigkeitsbereich von x endet und dessen Speicher gilt als frei. Wird nun die println-Funktion aufgerufen, würde dies zu undefiniertem Verhalten führen, da der Speicher auf den r zeigt nicht mehr gültig ist. Rust würde diesen Code ablehnen, dass es dieselbe Analyse wie oben gemacht hat und den Fehler erkannt hat. Hierfür wird das Konzept der Lifetimes verwendet. Rust weist hier jedem Gültigkeitsbereich (Scope) eine Lifetime zu, jede Variable, die in diesem Bereich gültig ist, hat auch diese Lifetime, sowie alle Referenzen, die auf diese Daten zeigen. In Listing 5 hat r die Lifetime des Gültigkeitsbereichs A und des inneren Bereichs B. x hat nur die Lifetime von B. Wird nun r im Gültigkeitsbereich A verwendet, wird überprüft welche Lifetime die Daten auf die r zeigt mindestens hat. In diesem Fall wäre das nur Bereich B aber nicht A und ist daher ungültig.

2.2 Foreign-Function-Interfaces in Rust

Die Credentialing-Entity ist Teil eines größeren Systems, welches in C/C++ geschrieben ist. Deshalb werden im Folgenden die Interoperabilität zwischen C- und Rust-Code über Foreign-Function-Interfaces(FFI) betrachtet. Hierbei wird auf das erstellen von Bindings eingegangen und nützliche Werkzeuge und Programme um diesen aufwendigen und fehleranfälligen Prozess zu vereinfachen und automatisieren:

2.2.1 C-Code aus Rust-Code aufrufen

Um eine C-Bibliothek in Rust verfügbar zu machen sind folgende Schritte notwendig:

- Bindings erzeugen

- Buildscript schreiben
- Sichere Abstraktion

Dieser Prozess lässt sich mit der Bindgen-Bibliothek später noch vereinfachen.

Bindings erzeugen:

Zunächst muss eine Rust-Repräsentation der C-Funktionen und Datentypen bereitgestellt werden. Soll für den Code-Ausschnitt in Listing 6 Bindings erstellt werden, sind einige Dinge zu beachten:

```
1  #include <stdint.h>
2
3  int sum(int a, int b);
4
5  typedef struct Point {
6      uint8_t x;
7      uint8_t y
8  }Point;
9
10 uint8_t get_len(struct Point* x);
```

Listing 6. Simples C-Code-Beispie

- Standarddatentypen in C haben hardwareabhängige variable Größe, Rust hat feste Größen
- Structs müssen interoperabel zwischen C und Rust sein
- In Rust werden Referenzen und keine Pointer verwendet

Der entsprechende Rust-Code der diese Probleme löst ist in der untenstehenden Listing 7 zu sehen:

```
1  use std::ffi::c_int;
2
3  #[repr(C)]
4  struct Point {
5      x: u8,
6      y: u8,
7  }
8
9  extern "C" {
10     fn sum(a: c_int, b: c_int) -> c_int;
11     fn get_len(x: *mut Point) -> u8;
12 }
```

Listing 7: Rust-Bindings

Hierin sieht man, dass Funktionssignaturen für C-Funktionen in einem "extern "C""-Block sein müssen. Um das Problem der variablen C-Datentapen zu lösen, bietet die Rust-Standardbibliothek eine Lösung an. Im "ffi"-Modul befinden sich

Definitionen, die die C-Datentypen zur Compilezeit mit den korrekten Datentypen ersetzt. Für die int-Variablen aus Listing 6 wird der c_int-Datentyp in Rust verwendet. Werden jedoch die Datentypen mit festen Größen aus dem stdint-Header verwendet, kann, wie im Point-Struct zu sehen, der entsprechende Rust-Datentyp verwendet werden.

Um C-Structs interoperabel zu machen, muss ein Rust-Struct mit dem repr(C)-Attribut versehen werden. Dieses sorgt dafür, dass der Rust-Compiler sich an Alignment-Regeln eines C-Compilers hält.

Sollen C-Funktionen verwendet werden können nicht einfach Rust Referenzen verwendet werden. Rust bietet hierfür einen Raw-Pointer-Datentyp an, dessen Umgang jedoch als "unsafe" angesehen wird, dass Rust hier nicht für Gültigkeit des Pointers garantiert.

<u>Sichere Abstraktion:</u>

C-Bindings werden in Rust grundsätzlich als Unsafe angesehen. Dies ist für den Nutzer der Bibliothek unpraktisch, da bei jeder Verwendung einer C-Funktion ein Unsafe-Block verwendet werden muss. Um dies zu verbessern, werden zu den direkten Bindings Wrapper-Funktionen geschrieben, welche eine sichere Abstraktion bieten. Soll zum Beispiel für die "get_len"-Funktion aus Listing 7 eine sichere Abstraktion erstellt werden besteht das Problem, das ein Raw-Pointer auf ein Point-Struct erwartet wird.

Da man in Rust aber aus Speichersicherheitsgründen mit Referenzen statt Pointern arbeitet, muss die Abstraktion eine Referenz nehmen und diese im Gültigkeitsbereich der Funktion in einen Raw-Pointer umwandeln.

Eine sichere Abstraktion für das Beispiel wäre:

```
1  pub fn get_point_len(x: &mut Point) -> u8 {
2      unsafe {
3          get_len(x as *mut Point)
4      }
5  }
```

Listing 8: Sichere Abstraktion

Wie in Listing 8 zu sehen, bietet Rust für Referenzen eine simple Typenumwandlung an.

Abschließend sollte erwähnt werden, dass durch diese Abstraktionen nur sichergestellt wird, dass nur gültige Werte und Pointer übergeben werden. Hat die

aufgerufene C-Funktion Fehler im Speichermanagement, ist ihre Benutzung weiterhin unsicher.

Buildscript:

Build-Scripte in Rust sind normale Rust-Programme, welche vor dem eigentlichen Kompilierungsprozess kompiliert und ausgeführt werden. Das Erzeugen eines Build-Scripts besteht aus zwei Teilen:

1. Optionaler Build der Bibliothek für die Bindings erzeugt werden sollen

Wenn eine bereits kompilierte Bibliothek vorliegt, kann dieser Schritt übersprungen werden. Rust bietet mehrere Möglichkeiten an um C-Libraries zu Kompilieren. Zum einen bietet die Rust Standard-Bibliothek die Möglichkeit Befehle auf der Kommandozeile auszuführen, was ermöglicht Shell-Scripte, Compiler und Buildsysteme wie CMake manuell aufzurufen. Zum anderen gibt es Crates, welche den Umgang mit dem nativen C-Compiler des Betriebsystems oder CMAKE erleichtern.

2. Dem rustc-Compiler mitteilen wo die zu linkende Bibliothek zu finden ist

In einem Buildscript kann Cargo Befehle gegeben werden, indem entsprechende Direktiven auf Standardout ausgegeben werden. Zeilen die mit "cargo:" beginnen werden in eine Datei umgeleitet, welche wiederum von Cargo vor der Kompilierung ausgelesen wird [12].

```
1  //Tell cargo the path of the directory of the library
2  println!("cargo:rustc-link-search=native=/path/to/library");
3  //Tell cargo the name of the library
4  println!("cargo:rustc-link-lib=static=name_of_library");
```

Listing 9: Cargo Bibliothekspfad mitteilen

Mit der zweiten Zeile in Listing 9 wird Cargo mitgeteilt, dass eine "native-Bibliothek" also C-Bibliothek im Pfad "/path/to/library" befindet. Mit der zweiten Zeile wird die Bibliothek als statische(static) mit dem Namen "name_of_library" benannt.

Bindgen:

Bindgen ist ein Softwaretool welches Rust-Bindings für C-Code generieren kann [13]. C++ wird mit Einschränkungen unterstützt [14]. Owner des Repositories ist die Rust-Foundation selbst[15].

Bindgen ist als CLI-Tool und als Bibliothek vorhanden, die in Rust-Buildscripts aufgerufen werden kann. Die Nutzung von Bindgen als Bibliothek innerhalb eines

Buildscripts ist die empfohlene Nutzungsmethode[16]. Für beide Methoden, werden die Header-Dateien der öffentlichen Schnittstellen benötigt. Sind mehrere Header vorhanden, wird empfohlen eine weitere Header-Datei zu erzeugen in der alle benötigten Header mit einer #include Direktive importiert werden. Nach Konvention wird dieser Header wrapper.h genannt.

Bindgen im Buildscript:

Bibliotheken die im Buildscript benötigt werden, müssen als Build-Dependecy zur Cargo.toml des Crates hinzugefügt werden. Im Buildscript muss Zunächst Bindgen konfiguriert werden. Hierfür wird eine Konfigurationsobjekt mit dem Builder-Pattern erzeugt. Eine minimale Konfiguration ist in Listing 10 dargestellt:

```
1  let bindings = bindgen::Builder::default()
2      // The input header we would like to generate
3      // bindings for.
4      .header("wrapper.h")
5      .clang_arg("-I../c_stuff/include")
6      // Finish the builder and generate the bindings.
7      .generate()
8      // Unwrap the Result and panic on failure.
9      .expect("Unable to generate bindings");
```

Listing 10: Bindgen konfigurieren

Mit der header()-Methode wird mitgeteilt wo sich der Header, für den Bindings erzeugt werden sollen, befindet. Zusätzlich können mit der clang_arg-Methode Compiler-flags an libClang übergeben werden, welches von Bindgen verwendet wird. In diesem Beispiel mithilfe der "-I"-Flag wird mitgeteilt, wo sich die in "wrapper.h" benannten Header befinden. Mit "generate" wird mithilfe der Konfigurationen versucht ein Binding-Objekt zu erzeugen.

Weitere Konfigurationsoptionen erlauben zum Bespiel mithilfe von Black- und White-lists einschränken für welche Funktionen, Typendefinitionen und Strukturen Bindings erzeugt werden oder zwingen Bindgen die Type-aliases aus der Core-Bibliothek zu verwenden anstatt Std, was für no_std environments nützlich ist(use-core).

Mithilfe dieses Binding-Objektes kann nun der Quellcode für die Bindings erzeugt werden und in einem wählbaren Verzeichnis der gespeichert werden.

Command-Line:

Das Bindgen CLI-Tool erlaubt es die gleichen Bindings mit den gleichen Konfigurationsmethoden zu erzeugen. Um die gleichen Bindings zu erzeugen wie im Abschnitt

Command-Line mit dem mit der Zusatzoption use-core kann folgender Befehl verwendet werden:

```
bindgen --use-core wrapper.h -o bindings.rs -- -I../c_stuff/include
```

Listing 11: Bindgen Kommandozeile

2.2.2 Rust-Code aus C-Code aufrufen

Um Rust-Code für die Verwendung in C bereit zu machen, sind folgende Schritte notwendig:

- Crate-Type ändern um gewünschten Bibliothekstyp zu erhalten (statisch, dynamisch)
- Funktionen und Structs C-kompatibel machen
- C-Header erstellen
  - Manuell
  - Cbindgen

Crate-Type anpassen:

Rust-Code kann in verschiedene Formen wie zum Beispiel statische und dynamische Bibliotheken oder Biniaries kompiliert werden. Um zu bestimmen in welcher Form kompiliert wird, kann eine Option in der "Cargo.toml"-Konfigurationsdatei hinzugefügt werden.

```
1  [lib]
2  crate-type = ["cdylib"]
3  #Or
4  [lib]
5  crate-type = ["staticlib"]
```

Listing 12: Crate-Type konfiguriern

Für die Interoperabilität mit C-Code werden die Typen "cdylib" für dynamisch geladene Bibliotheken und "staticlib" für statische Bibliotheken verwendet.

C-kompatibel machen:

Um dem Rust Compiler zu zeigen, dass eine Funktion C-kompatibel sein soll, wird diese mit dem Zusatz "pub extern "C" fn" versehen (siehe Listing 13).

```
1  #[no_mangle]
2  pub extern "C" fn important_operation(x:u32) -> bool{/*...*/}
```

Listing 13: Beispiel C-kompatible Rustfunktion

Mit dem "#[no_mangle]"-Macro wird das Rust Name-Mangling für diese Funktion ausgeschaltet [17].

Um Structs kompatibel zu machen sind weitere Maßnahmen notwendig:

Zum einen muss dafür gesorgt werden, dass sich der Rust Compiler an das C-Memory-Layout hält.

```
1  #[repr(C)]
2  struct Point { x: i32, y: i32,}
```

Listing 14: Rust Struct mit C-Memory-Layout

Dies wird durch das "#[repr(C)]"-Makro erreicht [18], wie in Listing 14 zu sehen.

Weiterhin werden Struct-Methoden von C nicht untestützt. Für diese muss eine Wrapper-Funktion erstellt werden (siehe Listing 15).

```
1  impl Point {
2      pub fn add_to_x(&mut self, x: i32,) {/*...*/}
3  }
4  //C-wrapper for add_to_x-method
5  pub extern "C" fn point_add_to_x(point: *mut Point, x: i32)
6  {/*...*/}
```

Listing 15: Wrapper für Struct-Methode

Beachtet muss hierbei werden, dass keine Referenzen verwendet werden dürfen, sondern stattdessen sogenannte Rawpointer verwendet werden müssen.

C-Header:

Damit gegen die Bibliothek gelinkt werden kann, wird eine Header-Datei benötigt. Dieser Header beinhaltet alle benötigten Funktions- und Typendefinitionen. Für einfache Beispiele ist es möglich den Header manuell zu schreiben.

Für die Struktur und die Funktionen aus Listing 14 und Listing 15:

```c
1  #include <stdint.h>
2  typedef struct Point
3  {
4      int32_t x;
5      int32_t y;
6  };
7
8  void point_add_to_x(struct Point *this, int32_t: x);
```

Listing 16: passender C-Header

Da Rust Standarddatentypen im Gegensatz zu C-Datentypen eine feste Größe haben, ist die Verwendung der Typendefinitionen des "stdint"-Headers ratsam, um Plattformunabhängig kompatibel zu bleiben.
Ist es nicht notwendig, dass der Nutzer der C-Bibliothek Zugriff auf die Felder erhält und nur mit Pointern auf Daten, die von der Rust-Bibiliothek erzeugt wurden, ist es ausreichend die Struct ohne Felder folgendermaßen zu definieren.

```c
1  typedef struct Point;
```

Listing 17: Struct ohne Felder

cbindgen:

Steigt die Komplexität der Bibliothek die Bindings erhalten soll, bietet es sich an diesen Prozess zu automatisieren. Hierfür gibt es das Tool cbindgen. Cbindgen ermöglicht das Erzeugen von C/C++-Headern für Rust-Bibliotheken mit einem C-Interface[19]. Im Gegensatz zu bindgen, dem Tool um Rust-Bindings für C-Bibliotheken zu erstellen, ist cbindgen kein offizielles Projekt der Rust-Foundation.

Genutzt werden kann das Tool entweder als CLI-Tool oder als Bibliothek in einem Buildscript. Vom Autor wird die Verwendung über die Kommandozeile empfohlen[20]. Dies geschieht mit folgendem Befehl:

```
cbindgen --config config.toml --crate my_lib --output bindgen.h
```

Listing 18: cbindgen Kommandozeile

Dieser Befehl verwendet drei Flags zur Konfigurierung.

1. --config

Cbindgen hat zahlreiche Konfigurationsmöglichkeiten, wie zum Beispiel die Möglichkeit den generierten Header in C, C++, oder Cython generieren zu lassen.

Sollen Konfigurationsmöglichkeiten vom Standard abweichen, kann eine Konfigurationsdatei erstellt werden. Eine vollständige Liste aller Optionen kann dem ReadMe im offiziellen Repository entnommen werden[21].

2. --crate

Befindet sich die Bibliothek in einem Cargo-Workspace mit mehreren Bibliotheken kann über diese Flag angegeben werden welche Bibliothek eine Header erhalten soll.

3. --output

Hiermit kann der Pfad und Dateiname der generierten Headerdatei angegeben werden.

2.3 Nützliche Rust-Konzepte

Rust ist unter Nutzern eine sehr beliebte Programmiersprache [22]. Dies lässt sich allerdings nicht ausschließlich auf die Speichersicherheitsgarantien zurückzuführen. Rust bietet auch einige Sprachkonzepte die die Softwareentwicklung vereinfachen. Dieses Kapitel beschäftigt sich mit einigen dieser Konzepte, die sich in diesem Projekt als nützlich erwiesen.

2.3.1 Rust-Enum

Enums in Rust teilen sich zwar den Namen mit dem gleichnamigen Sprachkonzept in C, sind aber wesentlich mächtiger, da sie als sogennte tagged-Unions agieren. Diese bieten wie in Zeile 2 und 3 in Abbildung 1 zu sehen, jeder Variante eines Enums Daten zuzuordnen. Für eine einfache Auswertung bietet Rust Werkzeuge wie das Match-Statement in den Zeilen 7-13. Dieses erzwingt zum einen eine vollständige Abarbeitung der Varianten, wobei wie in Zeile 12 zu sehen ein Unterstrich als Platzhalter für alle fehlenden Varianten dienen kann. Enthält eine Variante Daten wie zum Beispiel StateB in Zeile 9, kann dem Inhalt ein Variablenname gegeben werden. Mit diesem Namen kann innerhalb des entsprechenden Arms des Match-Statements auf die Daten zugegriffen werden.

```
1  enum Example {
2      StateA,
3      StateB(associatedData),
4      StateC(associatedData),
5  }
6  /*...*/
7  match example {
8      Example::StateA => todo!(),
9      Example::StateB(data) => {
10         todo!("Important stuff with data")
11     }
12     _ => todo!("All other cases"),
13 }
```

Abbildung 1: Rust-Enum

Rust-Enums eignen sich zum Beispiel um Zustandsmaschinen darzustellen, da hier nicht nur der Zustand selbst, sondern auch die Zustandsspezifischen Daten Teil des Zustands sind.

3 Architektur

Um eine Rust-Implementierung der Credentialing-Entity zu erstellen, muss erst verstanden werden, wie diese CE funktioniert und im Gesamtsystems des CAN-Demonstrators wirkt.

Hierfür wird in folgenden Schritten vorgegangen:

- Betrachten des Demonstrator-Gesamtsystem und der Interaktion zwischen CE und IA-Komponente
- Analyse der Komponenten der CE und deren Struktur
- Analyse des Ablaufs des Programms

3.1 Aufbau und Struktur

Der CAN-Demonstrator besteht aus einer Credentialing-Entity und einer oder mehreren IA-Komponenten, welchen alle ein einen gemeinsamen CAN-Bus (siehe Abbildung 2) angeschlossen sind.

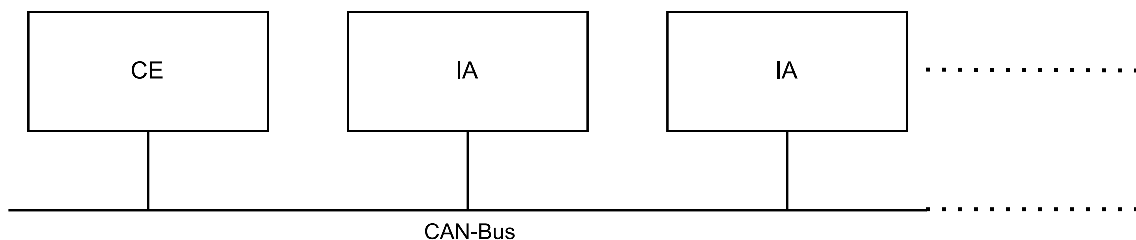

Abbildung 2: CAN-Demonstrator Grobstruktur

Um zu verstehen wie Nachrichten gesendet, empfangen werden, muss zunächst das innere der CE betrachtet werden. Betrachte man nun die untenstehende Abbildung 3 erkennt man vier Hauptkomponenten der Credentialing-Entity:

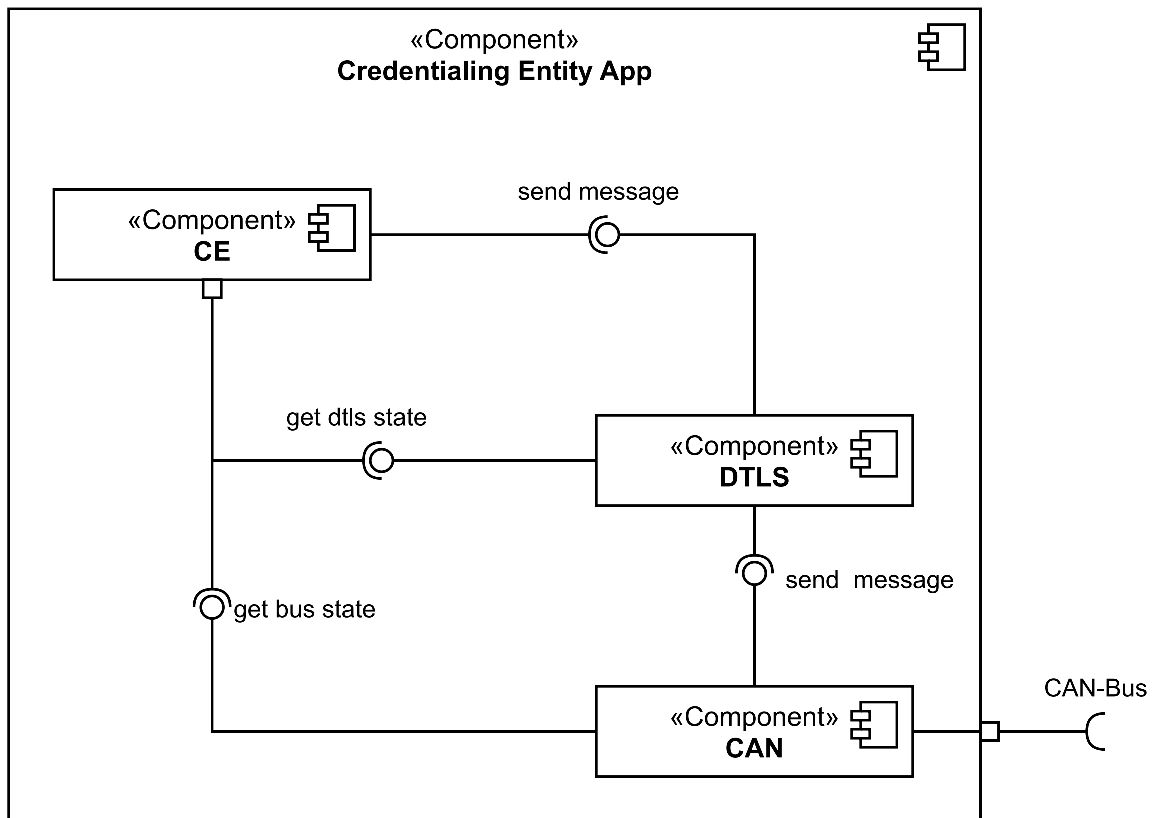

Abbildung 3: C++-Strukturdiagramm

1. CE-Bewirtschaftungs-Komponente

   Diese Komponente beinhaltet die Bewirtschaftungslogik. Um unabhängig von der verwendeten Bustechnologie zu bleiben zu können, ist die CE mit Vererbung umgesetzt (siehe Abbildung 4). Hierbei besitz die Basisklasse eine Instanz einer CA, eine öffentliche Funktion, um eine Runde des CE-Prozesses (siehe 3.2) abzuarbeiten und virtuelle Funktionen für die Buskommunikation. Eine konkrete Implementierung für diese Basisklasse ist im Demonstrator mit der CAN_OpenCE-Klasse vorhanden. Diese implementiert die virtuellen Funktionen. Zusätzlich besitzt ein die Basisklasse implementierende Klasse entweder eine Instanz des Busobjektes selbst oder wie im Falle der CAN-CE eine Objekt welches den Nachrichtenaustausch zum Kommunikationsprotokoll ermöglicht.

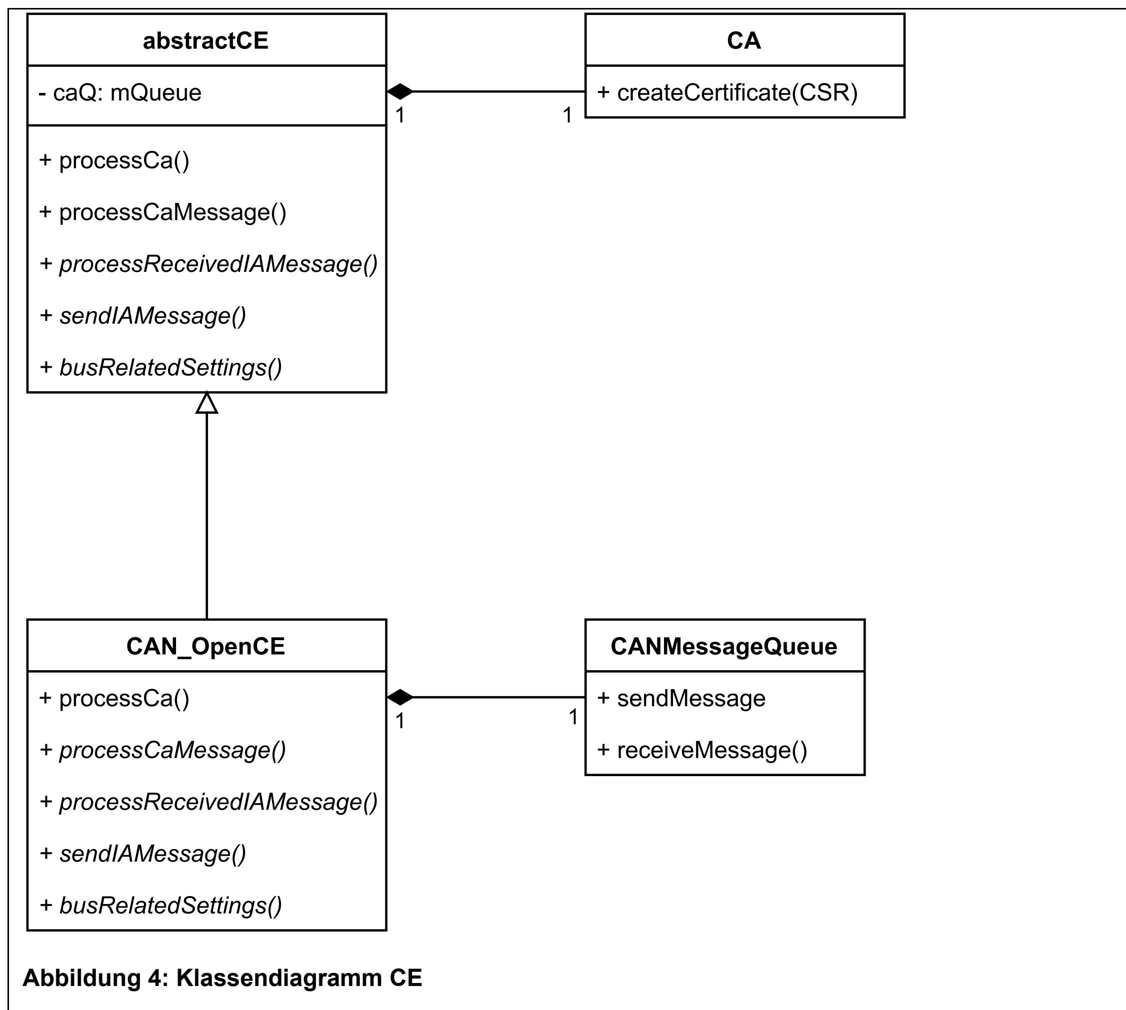

Abbildung 4: Klassendiagramm CE

2. CA-Komponente(intern)

   Dieser Komponente nimmt CSRs (Certificate Signing Request) entgegen, erstellt auf Basis darauf Zertifikate und schickt diese an die CE zurück. Sie ist in Abbildung 3 nicht abgebildet, da sie in der vorliegenden Implementierung intern in der CE-Komponente implementiert ist. Das sie aber auch als von der Credentialing-Entity getrennte Einheit vorliegen kann, wird sie aus Gründen der Vollständigkeit erwähnt.

3. DTLS-Komponente

   Diese Komponente kümmert sich um die Absicherung der CAN-Kommunikation mittels DTLS. Neben der Verarbeitung ein- und ausgehender Kommunikation, ermöglicht diese Schicht der CE Statusinformationen ausgehender DTLS-Verbindungen abzurufen und Einstellungen zu machen. Diese Einstellungen beinhalten die Einstellung des für DTLS-Kommunikation verwendeten Zertifikats und die Änderung des DTLS-Kommunikationsstatus.

4. CAN-Komponente
   Diese Komponente tauscht Nachrichten vom und zum physischen CAN-Bus aus unter der Verwendung des CAN-Open-Protokolls. Ähnlich wie die DTLS-Komponente ermöglicht auch die DTLS-Schicht Statusinformationen mit der CE auszutauschen.

Will die CE-Komponente nun eine Nachricht an eine IA-Komponente senden, fordert sie die DTLS-Komponente eine sichere Verbindung zur IA-Komponente aufzubauen. Ist dies Geschehen gibt die CE die zu versendende Nachricht an die DTLS-Schicht weiter. Diese gibt die Nachricht verschlüsselt an die CAN-Komponente weiter, welche die Nachricht im CAN-Open-konformen Paketen über den CAN-Bus versendet. Ankommende Nachrichten funktionieren umgekehrt. Die CAN-Komponente entpackt die Nachricht aus dem CAN-Open-Format aus und gibt diese an die DTLS-Schicht weiter. Diese Entschlüsselt die Pakete und reicht sie an die Credentialing-Entity weiter.

3.2 Logik und Ablauf

Der innere Ablauf des CE-Gesamtsystems lässt sich grob in vier logische Komponenten aufteilen, welche im Folgenden in der Reihenfolge von äußersten zur innersten Schicht betrachtet werden:

1. Ablauf in der Hauptschleife
2. Grobablauf innerhalb der CE-Komponente
3. Verbindungsaufbau zu IA-Komponenten
4. Logik zur Bewirtschaftung der IA-Komponenten

3.2.1 Hauptschleife

Die erste zu betrachtende Ebene der C++-Anwendung ist die Endlosschleife im Hauptprogramm. Wie in Abbildung 3 zu sehen besteht die Anwendung aus drei Komponenten. Die CE-, die DTLS- und die CAN-Komponente werden zunächst vor dem Eintritt in die Hauptschleife initialisiert. Im Anschluss wird die Endlosschleife betreten. In dieser werden die drei Komponenten jeweils einmal durchlaufen.

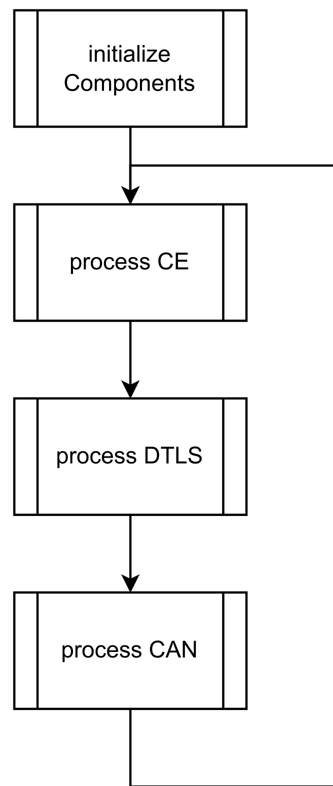

Abbildung 5: Hauptschleife C++-Programm

3.2.2 Grobablauf CE-Komponente

Als nächstes wird die CE-Komponente näher betrachtet. Hierbei wird zunächst der grobe Ablauf eines Durchlaufs der CE in der Hauptschleife, wie in Abbildung 6 dargestellt, betrachtet.

Zunächst werden Nachrichten von der internen CA empfangen und verarbeitet. Wird ein fertiges Zertifikat empfangen, wird es direkt an die entsprechende IA-Komponente verschickt. Im Anschluss werden die Nachrichten der IA-Komponenten vom CAN-Bus empfangen und verarbeitet.

Sind alle Nachrichten empfangen und verarbeitet worden, geht das Programm in einer Schleife über alle belkannten IA-Komponenten und setzt deren Bewirtschaftungsprozess fort. Hierfür wird zunächst überprüft, ob überhaupt eine Verbindung zur entsprechenden Komponente besteht. Ist gesicherte Kommunikation möglich, wird der aktuelle Bewirtschaftungsschritt fortgesetzt. Ist dieser Prozess für die aktuelle Komponente vollendet, werden nacheinander alle anderen Komponenten abgearbeitet.

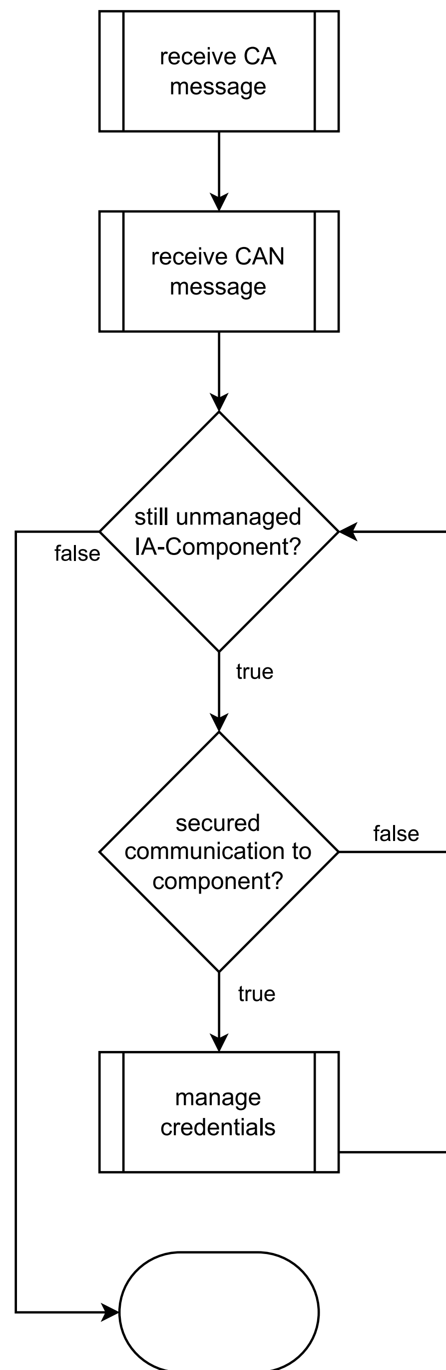

Abbildung 6: Flussdiagramm Grobablauf CE

Sind alle Komponenten abgearbeitet, ist ein Durchlauf der CE vollendet.

3.2.3 Verbindungsaufbau

Im vorigen Kapitel 3.2.2 wurde beschrieben, dass ein Verbindungsaufbau stattfinden muss, bevor eine IA-Komponente bewirtschaftet werden kann. Wie genau dieser Prozess abläuft wird jedoch nicht beschrieben und ist Gegenstand dieses Unterkapitels.

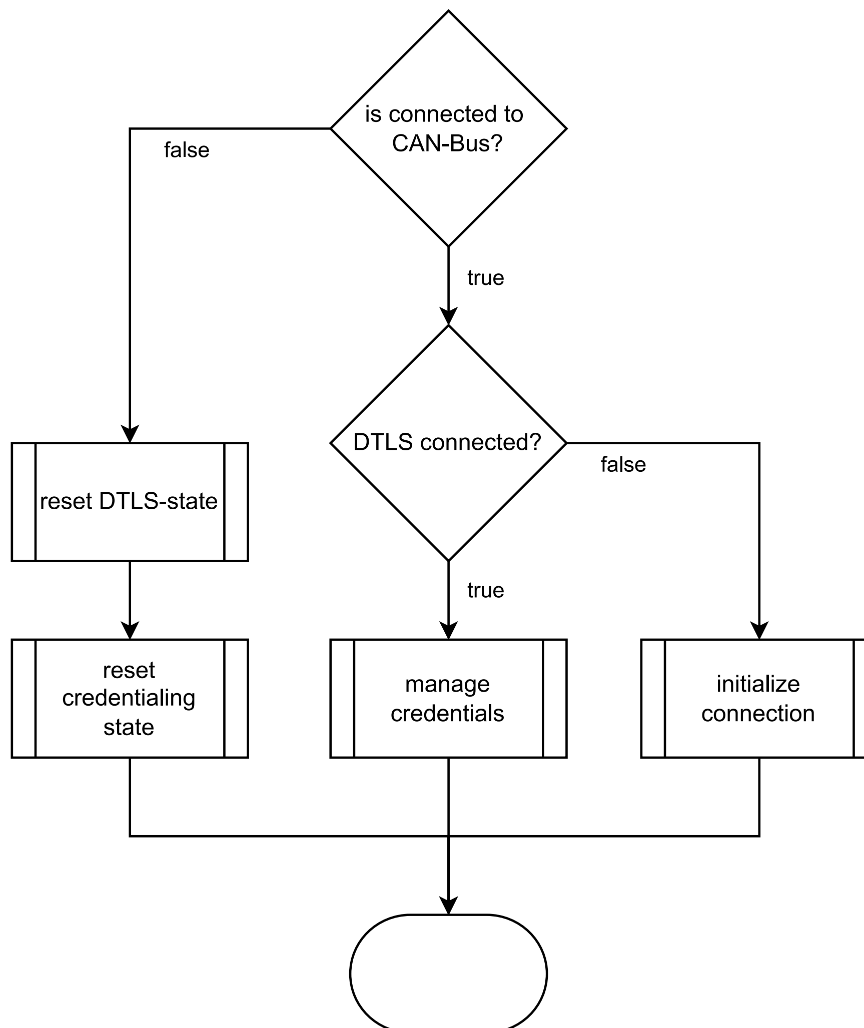

Abbildung 7: Flussdiagramm Verbindungsaufbau

In einem ersten Schritt wird bei der CAN-Komponente angefragt, ob die aktuell bewirtschaftete Komponente eine aktive Verbindung zum CAN-Bus hat. Ist keine Verbindung vorhanden, werden der Status der DTLS-Verbindung auf nicht verbunden und der Bewirtschaftungsstatus auf unbekannt gesetzt.

Ist die Komponente verbunden, wird überprüft, ob eine aktive DTLS-Verbindung besteht. Besteht noch keine Verbindung, wird der Verbindungsprozess in Gang gesetzt. Ist jedoch schon eine gesicherte Verbindung vorhanden, kann der Bewirtschaftungsprozess begonnen oder fortgesetzt werden.

3.2.4 Ablauf Bewirtschaftung

Ist eine gesicherte Verbindung zu einer IA-Komponente aufgebaut worden, kann der Bewirtschaftungsprozess beginnen. Wie dieser Prozess im Erfolgsfall aussehen kann, ist in der untenstehenden Abbildung 8 dargestellt.

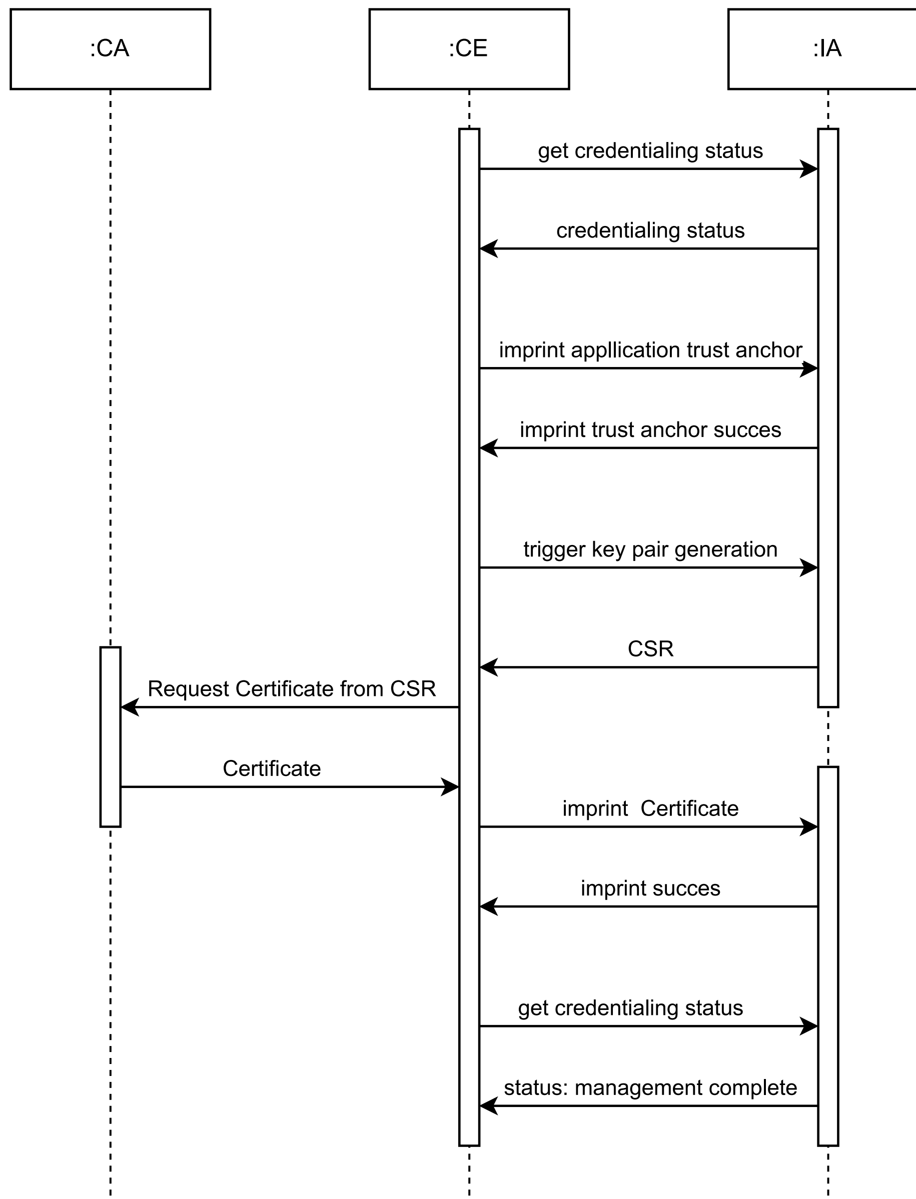

Abbildung 8: Sequenzdiagramm: Bewirtschaftung

Der Prozess, der kann in 4 Schritte eingeteilt werden:

1. Bewirtschaftungsstatus abfragen

   Hierbei wird abgefragt, für welche Zertifikatsdomänen bereits Zertifikate vorhanden sind. Im dargestellten Fall hat die Inbesitznahme bereits Erfolg gehabt und die Komponente besitzt bereits Hersteller- und

Anwendungszertifikate. Dies teilt die Komponente in einer Antwort der CE mit.

2. Vertrauensanker einprägen

   In diesem Schritt wird der Komponente das Wurzelzertifikat für die Anwendungsebene gegeben. Die IA-Komponente bestätigt dies mit einer Erfolgsmeldung als Antwort.

3. Zertifikat erstellen und aufbringen

   Dieser Schritt kann in 3 Unterschritte aufgeteilt werden. Zuerst wird die IA-Komponente aufgefordert ein Schlüsselpaar zu erzeugen und mit diesen Schlüsseln einen CSR zu erzeugen. Diesen schickt die Komponente an die CE.

   Im nächsten Schritt wird übergibt die Credentialing-Entity den CSR der CA. Die CA erzeugt das Zertifikat und gibt dieses der CE.

   Im letzten Schritt übergibt die CE das erzeugte Zertifikat der IA-Komponente und diese erwidert mit einer Erfolgsmeldung.

4. Status erneut abfragen

   Im letzten Schritt wird der Zertifikatsstatus der Komponente erneut abgefragt, um die Erfolgreiche Bewirtschaftung zu bestätigen.

4 Bibliotheken

Um die Credentialing-Entity umzusetzen, werden kryptografische Funktionalitäten gebraucht. Für solche ist es sinnvoll auf bereits getestete und erprobte Software-Bibliotheken zurückzugreifen. Dieses Kapitel beschäftigt sich zunächst mit den Funktionalitäten für die auf Bibliotheken zurückgegriffen werden. Im Anschluss werden diverse Bibliotheken, die gewünschte Funktionalitäten implementieren, genauer untersucht. Abschließend werden Bibliotheken zur Verwendung ausgewählt.

Die CE benötigt an zwei Stellen solche Funktionalitäten. Es wird DTLS benötigt für die abgesicherte Kommunikation über den CAN-Bus benötigt, zum anderen wird für die interne CA die Fähigkeit benötigt Zertifikate auf Basis von Certificate-Signing-Requests (CSR) benötigt. Für DTLS gibt es folgende Anforderungen:

- DTLS

  Der CAN-Bus bietet keine vollumfängliche Zuverlässlichkeit an. Reguläres TLS wie es in zum Beispiel TCP-Verbindungen verwendet wird, erwartet aber eine zuverlässige Verbindung. Für unzuverlässige Übertragungen gibt es eine Variante die neben Authentifizierung und Verschlüsselung der Kommunikation auch für Zuverlässigkeit während des Handshakes sorgt. Diese wird DTLS genannt.

- TLS 1.2 und 1.3

  Das TLS-Protokoll und das darauf aufbauende DTLS werden weiterentwickelt. Im Demonstrator wird die DTLS-Version 1.2 verwendet. Um zukünftig die Option zu haben, auf die neuere Version 1.3 umsteigen zu können wird auch überprüft, ob diese implementiert ist.

- TLS soll Certificate-Revocation-Lists(CRL) unterstützen

  Die CE des Demonstrators besitzt zurzeit noch keine noch keinen Mechanismus, um Zertifikate vor deren Ablaufdatum ungültig zu machen. Dies ist jedoch geplant und die Information über abgelaufene Zertifikate soll

mithilfe von CRLs verteilt werden. Deshalb muss die DTLS-Schicht abgelaufene Zertifikate mithilfe von CLRs erkennen können.

Um die CA-Funktionalität umzusetzen werden folgende Funktionen benötigt:

- Einlesen von X.509-Zertifikaten und Schlüsseln in DER- und PEM-codierter Form

  Das Zertifikat und der private Schlüssel mit dem die interne CA Zertifikate signiert liegt in PEM-codierter Form vor. Erzeugte Zertifikate liegen DER-codiert vor. Sollen diese überprüft werden, muss DER-Decodierung möglich sein.

- Auslesen der Zertifikatsfelder

  Bei der Überprüfung von Zertifikaten, können Informationen aus Zertifikatsfeldern wie zum Beispiel Zertifikats-Ids relevant sein. Um dies überprüfen zu können, muss die Bibliothek Felder auslesen und anzeigen können.

- Validieren von Zertifikaten

  Ist die CA extern, soll es möglich sein das erzeugte Zertifikat auf der CE zu überprüfen. Hierzu gehört die Validierung der Signatur.

- Lesen und schreiben von Certificate-Signing-Requests

  Die CA muss CSRs einer IA-Komponennte lesen können um auf Basis dieser Zertifikate erstellen zu können

- Erstellen von Zertifikaten

Für kryptografische Funktionalitäten ist es sinnvoll auf bereits getestete und erprobte Software-Bibliotheken zurückzugreifen. Für diese Bibliotheken gibt es neben den oben gewünschten Features noch weitere Faktoren, auf die bei der Auswahl geachtet werden soll:

- Keine Wrapper um C-Bibliotheken:

Das Ziel des Projekts ist eine Funktionalität in der Speichersicheren Sprache Rust umzusetzen. Für die sicherheitskritischen Teile des Programms eine C-Bibliothek zu verwenden ist daher nicht erwünscht.

- Evtl. #[no_std] kompatibel

Die no_std Option verhindert den Zugriff auf die Teile der Standardbibliothek, die Zugriff auf ein unterliegendes Betriebssystem benötigen, wie einen Heap-Allocator. Um sich die Möglichkeit freizuhalten, die Softwarekomponente auf einem Baremetal-Embedded-Device verwenden zu können, sind no_std-kompatible Bibliotheken zu bevorzugen.

- Aktivität im Repository/Projekt:

Bei Open Source Projekten kommt es oft vor, dass ein Projekt oft nur einer Person stammt und von dieser nicht mehr gepflegt wird. Wenn bei der Verwendung eines solches Projektes Fehler auftreten, muss die komplette Wartung selbst übernommen werden. Um dieses Risiko zu minimieren, sollten Softwarebibliotheken mit mehreren Besitzern gewählt werden. Kürzliche Aktivität im Repository ist ein weiteres wünschenswertes Kriterium.

- Freizügige Open-Source-Lizenz(MIT/Apache 2.0/ BSD-3)

Die Lizenzen, unter denen die verwendeten Bibliotheken veröffentlicht werden, sollten Endanwender nicht eingeschränken. Dies bedeutet, dass die verwendeten Lizensen einerseits nicht zwingt den Sourcecode aller abgeleiteten Werke zu veröffentlichen(Copyleft-Klausel) [23]. Zum anderen muss die Lizenz die kommerzielle Verwendung mit minimalen Einschränkungen erlauben. Solche Lizenzen werden als freizügige(permissiv) Open-Source-Lizensen bezeichnet [24]. Häufig verwendete Lizensen dieser Kategorie sind die MIT- die Apache-2.0- und die BSD-3-Lizenz [25].

4.1 TLS-Bibliotheken

Zunächst wurden TLS-Bibliotheken untersucht. Gesucht Es wurden hierbei Rusts offizieller Package-Index crates.io [26] und Github [27] durchsucht. Zunächst wurden alle Bibliotheken in die nähere Auswahl genommen, die eine reine Rust-

Implementierung von TLS bieten. Vier Bibliotheken wurden hierbei gefunden und im Folgenden die Funktionen für DTLS und CA untersucht.

4.1.1 Rustls

Rustls ist laut offizieller Beschreibung eine moderne TLS-Bibliothek geschrieben in Rust, die Produktionsreif ist und keine "breaking interface changes" mehr plant [28].

Rustls ist Produktionsreif, hat eine aktive Community und ermöglicht TLS-Kommunikation in den Versionen 1.2 und 1.3. Für den Schlüsselaustausch wird ECDH unterstützt, ECDSA und RSA für Authentifizierung, chacha20 und AES128/256gcm für Verschlüsselung und SHA256/384/512 als Hash-Algorithmus. DTLS und CRLs werden zur Zeit der Untersuchung nicht angeboten. *Rustls* liefert auch keine Werkzeuge um mit X509-Zertifikaten, CSRs zu interagieren.

4.1.2 Embedded-TLS

Embedded-TLS ist laut offizieller Beschreibung eine Rust-native Implementierung von TLS 1.3 das in einer No_STD-Umgebung funktioniert.

Die Software befindet sich noch im Entwicklungsstatus [29]. Zum Beispiel werden deswegen Zertifikate beim TLS-Verkehr nicht verifiziert und es wird nur eine Ciphersuite Aes128GcmSha256 mit ECDH angeboten. DTLS wird zum Untersuchungszeitpunkt nicht implementiert. Da die Bibliothek auch nur auf TLS fokussiert ist, werden Werkzeuge zum Umgang mit Zertifikaten nicht angeboten.

4.1.3 WebRTC-DTLS

Laut offizieller Beschreibung ist *WebRTC-DTLS* Teil einer Rust-nativen Implementierung des WebRTC-Stacks. Diese ist von der Go nativen Pion Implementierung abgeleitet [30].

WebRTC-DTLS bietet kein reguläres TLS nur DTLS in der Version 1.2. Es bietet keine Unterstützung für CRLs. Für Ciphersuites wird ECDH und PSK zur Verschlüsselung angeboten, ECDSA für Authentifizierung, für die Verschlüsselung AES128ccm, AES128gcm und AES256cbc, und SHA 256 als Hash-Algorithmus.

Erzeugung und Umgang mit Zertifikaten wird auch nicht angeboten. Die Bibliothek ist stark in den WebRTC-Stack integriert und daher nicht alleine verwendbar.

4.1.4 SaiTLS

SaiTLS ist bietet laut eigener Beschreibung eine Client- und Server-Side-Implementierung die auf smoltcp aufbaut [31].

Es bietet kein DTLS und keine Zertifikatsvalidierung in TLS. Angebotene Ciphersuites sind TLS_AES_128_GCM_SHA256, TLS_AES_256_GCM_SHA384, TLS_CHACHA20_POLY1305_SHA256und TLS_AES_128_CCM_SHA256. Auch werden keine Werkzeuge zum Umgang mit Zertifikaten bereitgestellt. Die Bibliothek ist fest auf smoltcp aufgebaut und daher nicht brauchbar für die Verwendung mit CAN.

4.1.5 Fazit

Das Rust Ökosystem, bietet zum Untersuchungszeitpunkt keine Bibliothek an, die den Anforderungen entspricht. Die einzige Bibliothek die DTLS(WebRTC-DTLS) implementiert ist zu stark in den WebRTC-Stack integriert, um für den CAN-Demonstrator verwendet zu werden. Auch bietet keine Bibliothek Werkzeuge zum Erstellen und Arbeiten mit Zertifikaten, wie zum Beispiel das im Demonstrator verwendete mbedTLS. Eine vollständige Übersicht befindet sich im Anhang 10.2.

4.2 X.509-Zertifikats-Bibliothek

Da keine der in 4.1 untersuchten Bibliotheken die für die CA benötigten Funktionalitäten liefert, müssen im nächsten Schritt nach Bibliotheken für die CA gesucht werden.

Hierfür wurde wieder der offizielle Rust-Package-Index crates.io durchsucht. Gesucht wurde nach den Schlagworten "Certificate" und "X509". Hierbei wurden zunächst Projekte gefiltert die laut Crates.io keinen Release im letzten Jahr hatten. Folgende sieben Projekte wurden hierbei gefunden und untersucht.

4.2.1 rpki

Die rpki-Bibliothek ist laut Beschreibung eine Bibliothek zum valideren und erzeugen von Resource-Public-Key-Infrastructure(RPKI)-Daten und unterstützt für diesen Zweck Zertifikate und CSRs [32].

Das bedeutet, dass zum Beispiel nur Zertifikate mit dem RPKI-Profil nach RFC 6487 mit dieser Bibliothek funktionieren[33]. Zertifikate erstellen ist auch nicht Teil des Funktionsumfang dieser Bibliothek. Daher ist sie ungeeignet.

4.2.2 X509-cert

X509-cert bietet laut Beschreibung Encoder und Decoder-Funktionalität für X.509-Zertifikate nach RFC5280[34].

Es werden das Lesen von Zertifikaten, CSRs und CRLs angeboten. Das Erstellen und Verifizieren fehlten zum Zeitpunkt der Überprüfung, sind jedoch geplante Features. Interessant an dieser Bibliothek ist die no_STD-Kompatibilität, wenn auf RSA-Verschlüsselung verzichtet wird. Die Bibliothek ist zum Betrachtungszeitpunkt ungeeignet aufgrund fehlender Funktionen, aber kann in der Zukunft eine gute Alternative werden.

4.2.3 barebones-x509

barebones-x509 bietet laut eigener Beschreibung ein Low-Level-Interface zum Verifizieren von X.509 Zertifikaten [35].

Der Bibliothek fehlen jedoch Unterstützung für CSRs und CRLs und die Erstellung von Zertifikaten. Auch gibt es keine Aktivität im Repository. Diese negativen Eigenschaften machen die Bibliothek unbrauchbar für die Verwendung im Demostrator.

4.2.4 x509-certificate

x509-certificate bietet laut eigener Beschreibung die Möglichkeit X.509 Zertifikate im BER, DER und PEM-Format zu parsen und die Zertifikate zu validieren [34].

Sie bietet das Lesen, Schreiben von Zertifikaten, CSRs und CRLs an. Auch die Verifizierung von Signaturen und Zertifikatsketten wird angeboten. Ohne Heap funktioniert diese Bibliothek nicht. Ein Problem stellt jedoch die Lizenz dar. Die Mozilla-Public-License(MPL) 2.0 bringt zusätzliche Forderungen an kommerzielle Nutzer der Bibliothek im Vergleich zu den anderen Freizügigen Lizenztypen.

4.2.5 picky

picky ist eine Sammlung von Libraries die sich mit verschiedenen PKI-verwandten Themen beschäftigen [36].

Hierzu zählen unter anderem das Parsen, Validieren und Verifizieren von X.509-Zertifikaten. Es sind lesen und schreiben von Zertifikaten und CSRs angeboten, allerdings werden mit Signaturen nur mit RSA und nicht ECDSA unterstützt. CRLs und no_std werden auch nicht unterstützt. Da das Zertifikatsprofil von FieldPKI ECDSA fordert ist diese Bibliothek nicht brauchbar.

4.2.6 X.509 Parser

Diese Bibliothek beschreibt sich selbst als einen X.509-v3-Parser der auf dem nom-Parser-Combinator-Framework [37].

Es bietet das Einlesen von Zertifikaten, CSRs und CRLs und deren Validierung an. Das Schreiben von Zertifikaten wird jedoch nicht angeboten und die Bibliothek ist nicht no_std-kompatibel. Durch die fehlende Möglichkeit Zertifikate zu erstellen, ist diese Bibliothek nicht für die CA verwendbar.

4.2.7 rcgen

Eine Bibliothek mit simplem Interface um X509-Zertifikaten zu erzeugen [38].

Diese Bibliothek baut auf 4.2.6 X.509 Parser auf. Es bietet daher das Lesen von Zertifikaten, CSRs und CRLs an. Zusätzlich ist die Erstellung von CSRs und Zertifikaten möglich, wobei sowohl ECDSA und RSA als Signaturalgorithmen unterstützt werden. Validierung von Zertifikaten ist direkt nicht möglich. No_std-Kompatibilität ist auch nicht gegeben.

4.2.8 Fazit

Im Rust-Ökosystem gibt es einige Bibliotheken, welche sich mit X509-Zertifikaten beschäftigen. X509-cert sieht vielversprechend aus und bietet als einzige der untersuchten Bibliotheken no_std an. Durch die fehlen Validierungsfunktion kann sie zum Untersuchungszeitpunkt nicht verwendet werden, da diese Funktion geplant ist, kann sie in Zukunft ein guter Ersatz werden. x509-certificate bietet alle nötigen Funktionen ist aber durch die MPL-Lizenz nicht in diesem Projekt brauchbar.

Die Kombination aus X.509 Parser und der darauf aufgebauten Bibliothek rcgen ist aktuell die einzige Kombination, die alle geforderten Features benötigt.

5 Rust-Architektur

Um die in Kapitel 3.1 beschriebene Struktur in Rust umzusetzen muss zunächst deren Aufbau und Struktur beschrieben werden. Dieses Kapitel beschäftigt sich zunächst mit den Schwierigkeiten der Portierung von C++ zu Rust. Im Anschluss wird auf die Grobarchitektur des Demonstrators mit der Rust-CE eingegangen. Zum Schluss wir auf die konkrete Architektur der Rust-Komponente eingegangen.

5.1 Schwierigkeiten und Probleme bei der Portierung

Bei der Entwicklung der Rust-Architektur wurde die C++-Variante als Vorbild genommen. Diese kann aber nicht eins zu eins übernommen werden. Gründe hierfür sind:

- Fehlende/ unterschiedliche Konzepte in den Programmiersprachen

  Die C++-Implementierung verwendet wie im Kapitel Aufbau und Struktur beschrieben Klassen und Vererbung und die Technologieunabhängigkeit zu erreichen. Rust besitzt keine Klassen und Vererbung im klassischen Sinne[39], sondern nur Interfaces, hier Traits genannt, um Polymorphismen zu implementieren.

- Fehlende Funktionalitäten/Bibliotheken

  Wie in der Bibliotheksrecherche in Kapitel 4.1 festgestellt wurde, gibt es keine passende Bibliothek um DTLS umzusetzen. Deshalb wird auf die DTLS-Komponente aus der C++-Implementierung zurückgegriffen. Auch für den CAN-Open-Kommunikationsstapel wird die bereits vorhandene Komponente aus dem Demonstrator verwendet.

- Kommunikation über ein Foreign-Function-Interface

  Da der Demonstrator mit der Rust-CE in zwei verschiedenen Programmiersprachen geschrieben ist, muss der Kommunikation zwischen den Teilen in verschiedenen Programmiersprachen besondere Aufmerksamkeit geschenkt werden. So sollten Schnittstellen auf das Notwendige reduziert werden, und Datenaustausch möglichst auf primitive

Datentypen reduziert werden, um Memorylayout-Probleme von Grund auf auszuschließen.

5.2 Grobarchitektur Gesamtsystem

Betrachtet man den Aufbau des Credentialing-Entity-Gesamtsystems mit dem Rust Teil, dargestellt in Abbildung 9, sind einige Aspekte zu beachten.

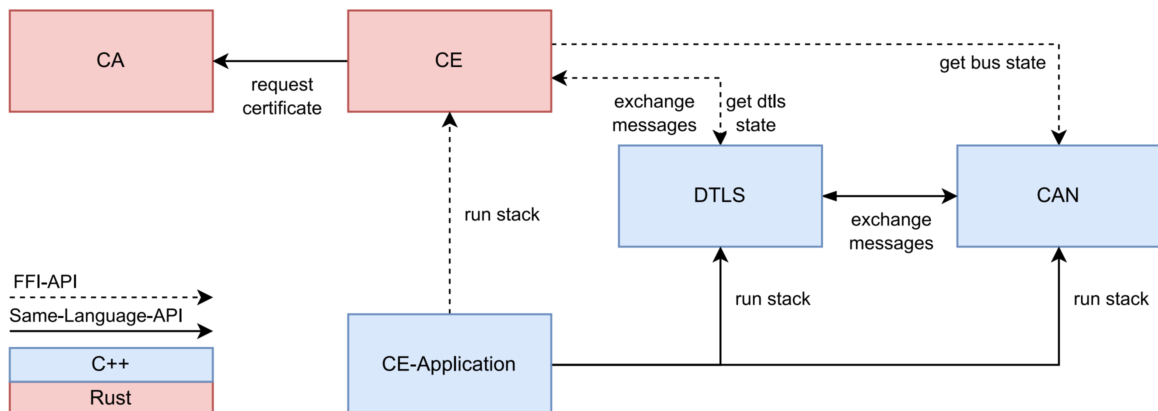

Abbildung 9: Grobarchitektur Gesamtsystem Rust

- CA-Komponente separiert

  Obwohl die CA-Komponente genauso wie in der C++-Implementierung als interne CA implementiert ist, wird sie in Abbildung 9 als separate Einheit dargestellt. Grund hierfür sind die Ergebnisse der Bibliotheksrecherche in Kapitel 4.2. Die gewählten Zertifikatsbibliotheken sind nicht no_std-kompatibel. Da alle anderen Komponenten der CE jedoch ohne Heap-Speicher auskommt, wurde alle Zertifikatsrelevanten Funktionen in die CA geschoben und als separates Modul gekapselt.

- Rust-Anteil

  Wie in 5.1 beschrieben ist nur die CE- und die zugehörige CA-Komponente in Rust geschrieben. Der Kommunikationsstapel aus CAN und DTLS, sowie die Anwendung selbst sind in C geschrieben.

- Bidirektionale Kommunikation über Foreign-Function-Interface

  Der Fall, dass Anwendung die die CE-Komponente aufruft, ist der Trivialfall für Foreign-Function-Interfaces. Von der Rust-Bibliothek hingegen Informationen zur C-Anwendung zu bringen, stellt hingegen eine

Herausforderung dar. In diesem Fall werden Rückruffunktionen zum Abrufen der Verbindungsstatus und dem bidirektionalen Austausch von Nachrichten verwendet.

5.3 Architektur Rust-CE

Aus den in 5.1 beschriebenen Herausforderung und den Anforderungen an das System, wurde die in der untenstehenden Abbildung 10 dargestellten Architektur umgesetzt.

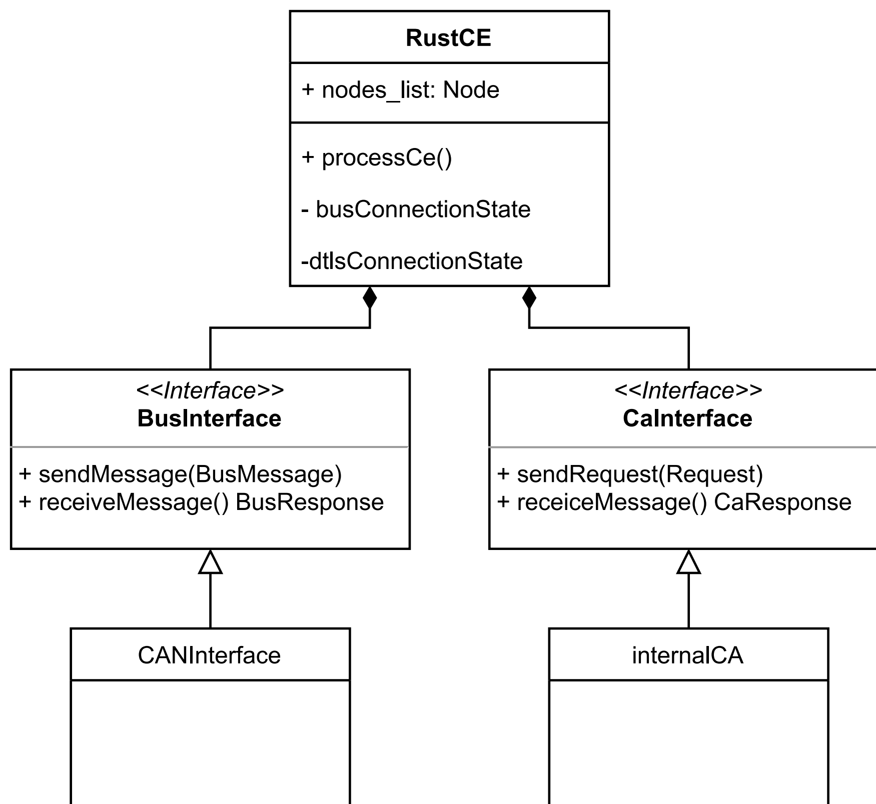

Abbildung 10: Klassendiagramm Rust-CE

Die Rust-Programmiersprache verzichtet bewusst auf Vererbung komplexe Klassenhierarchien zu vermeiden. Um dennoch Varianten wie zum Beispiel die Unterstützung verschiedener Bussysteme zu ermöglichen gibt es die zuvor erwähnten Traits. Diese ermöglichen Polymorphie nach dem Designprinzip Komposition über Vererbung. Aus diesem Grund besteht die Rust-CE aus je einem Objekt, die das Bus-Interface und das CA-Interface implementieren. Neben dieser Komposition besteht die Rust-CE sonst noch aus einer Liste, welche die Informationen über bewirtschaftete IA-Komponenten. Zu Letzt besitzt die Rust-CE

die Funktionen um die Rückruffunktionen für Verbindungsstatusinformationen aufzurufen.

Die öffentliche Schnittstelle besteht analog zur C++ Implementierung nur aus einer Funktion, die einmal den Bewirtschaftungsprozess durcharbeitet.

6 Umsetzung

Das folgende Kapitel beschäftigt sich mit der Umsetzung der in Kapitel 5 beschriebenen Architektur. Hierbei wird auf wichtige Schnittstellen und Probleme bei der Entwicklung der jeweiligen Komponenten eingegangen. Folgende Komponenten werden hierbei näher betrachtet:

- no_std-kompatibler Ringpuffer
- CA
- CE

Als letztes wird die Integration und Verifikation der CE beleuchtet.

6.1 Ringpuffer

Ringpuffer als Komponente wurden nicht in der in Kapitel 5 beschriebenen Architektur erwähnt, obwohl sie dennoch benötigt werden. An einigen Stellen wie zum Beispiel der Schnittstelle zwischen CE und CA wird ein asynchroner Ablauf vorausgesetzt. Nachrichten und Daten zwischen solchen Schichten werden in einem Pufferspeicher zwischengespeichert. Hierfür sollen Warteschlangen verwendet werden, um mehrere Nachrichten im Puffer zu ermöglichen und eine Abarbeitung in Sendereihenfolge zu garantieren. Eine gängige Implementierung einer Solchen Warteschlange in Umgebungen ohne dynamische Speicherreservierung ist der Ringpuffer, da diese eine feste Größe haben. Da es allerdings keine Ringpuffer-Implementierung im Rust Standard gibt, muss eine eigene Variante bereitgestellt werden. Folgende Aspekte werden in diesem Kapitel beleuchtet:

- Wie funktioniert ein Ringpuffer
- Besonderheiten dieser Implementierung

6.1.1 Wie funktioniert ein Ringpuffer

Ein Ringpuffer besteht aus einem Array als Puffer für die Daten, einem Lesekopf und einem Schreibkopf (siehe Abbildung 11). Der Schreibkopf zeigt auf den nächsten Platz im Array an den geschrieben werden kann und der Lesekopf zeigt

auf den nächsten Platz, der noch nicht ausgelesen wurde. Wird gelesen, wandert der Lesekopf einen Platz weiter. Das Äquivalente passiert beim Schreiben mit dem Schreibkopf. Diese Struktur erklärt dennoch nicht, warum die Struktur Ringpuffer genannt wird. Hierfür muss der Fall betrachtet werden, dass der Schreibkopf auf den letzten Platz des Arrays zeigt. Wird nun in diesen letzten Platz geschrieben, muss der Schreibkopf weitergehen. Da er aber schon im letzten Platz ist, wandert er zum Anfang des Arrays. Die beiden Enden, in Abbildung 11 in Rot dargestellt, werden also zu einem Ring zusammengenäht.

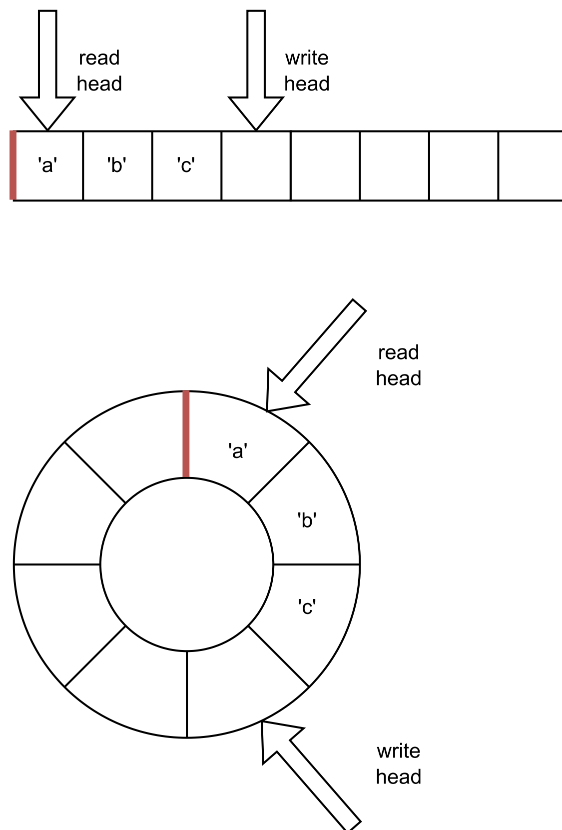

Abbildung 11: Funktionsprinzip Ringpuffer

6.1.2 Implementierung

Um die Besonderheiten der Ringpufferimplementierung zu verstehen, betrachte man zunächst die Struct-Deklarierung in Listing 19.

```
1  pub struct MessageQueue<const SIZE: usize> {
2      buf: [u8; SIZE],
3      read_head: usize,
4      write_head: usize,
5  }
```

Listing 19: Rinbuffer-Struct

Auffällig ist zunächst der generische Parameter " const SIZE: usize " in Zeile eins, der dann auch für die Länge des Arrays in Zeile zwei verwendet wird. Hierbei handelt es sich um einen sogenannten "const generic"-Parameter. Dieser Parameter ermöglicht Ringpuffer verschiedener Länge zu erzeugen, die aber

dennoch auf dem Stack liegen können, da der konstante Parameter und damit die Länge des Arrays während des Kompilierprozesses schon feststehen müssen.

Ringpuffer sind meist so implementiert, dass jeweils Elemente fester Größe in den Puffer geschrieben werden. Sollen aber zum Beispiel Nachrichten mit stark variierender Größe versendet werden, wäre es sehr ineffizient für jede Nachricht Platz für die längstmögliche Nachricht zur Verfügung zu stellen. Da aber Nachrichten mit stark variierender Länge in der CA-Schnittstelle verwendet werden, muss eine Möglichkeit gefunden werden, den Prozess effizienter zu machen.

```
1  pub fn push(&mut self, msg: &[u8]) -> bool { /*...*/}
2  pub fn pop(&mut self, buf: &mut [u8]) -> Option<usize> { /*...*/}
```

Listing 20: Funktionssignatur Ringpuffer push pull

Die hier verwendete Lösung ist eine Ringpufferschnittstelle (siehe Listing 20), die erwartet, dass die Nachrichten als Bytesequenz serialisiert übergeben wird.

Ein Problem gilt es hierbei jedoch zu beachten: Woher weiß der Ringpuffer beim Lesen wie viele Bytes gelesen werden müssen. Hierfür wird vor die eigentliche Nachricht zwei Bytes mit der Länge geschrieben (siehe Abbildung 12). Diese Konstellation lässt eine maximal Nachrichtenlänge von $2^{16} - 1 = 65535$ Bytes zu.

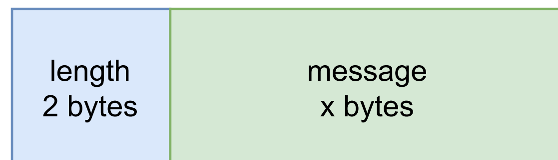

Abbildung 12: Aufbau Ringpuffer-Nachrichten

6.2 CA

Dieses Unterkapitel beschäftigt sich mit der Credentialing-Authority, welche die Zertifikate für die bewirtschafteten IA-Komponenten erzeugt. Hierbei wird auf folgende Aspekte der Implementierung eingegangen:

- CA-Trait (Interface)
- Implementierung der internen CA

6.2.1 CA-Trait

Um verschiedene austauschbare CA-Implementierung zu ermöglichen, wurde in den Architekturüberlegungen in Kapitel 5.3 ein Interface ausgewählt, um dies zu erreichen. Man betrachte die Implementierung dieses Traits ist in Listing 21.

```
1  pub trait Ca {
2      fn send(&mut self, req: Request) -> Option<usize>;
3      fn receive(&mut self, buf: &mut [u8]) -> Option<Response>;
4  }
5
6  pub enum Request<'a> {
7      GetCertificate(NodeId, Domain, Csr<'a>),
8  }
9
10 pub enum Response {
11     RequestDenied(NodeId),
12     GetCertificate(NodeId, Domain, usize),
13     InvalidMessage,
14 }
```

Listing 21: CA-Trait und zugehörige Datentypen

Der Trait selbst besteht aus zwei Funktionen:

1. send-Funktion

   Diese Funktion nimmt einen Request entgegen, welcher, wie in Zeile sechs zu sehen, alle notwendigen Daten für die zu versendende Nachricht enthält. Als Rückgabewert wird eine Rust-Option verwendet, ein enum aus dem Standard, welcher entweder eine Instanz des in den "<>" umschlossenen Datentyp enthält, oder nichts. Hier wird im Erfolgsfall die Menge der geschriebenen Bytes zurückgegeben oder Nichts im Fehlerfall.

2. receive-Funktion

   Diese Funktion nimmt einen Puffer entgegen, in dem empfangene Nachrichten kopiert werden und liefert eine Option zurück, die im Erfolgsfall eine Instanz des Response-Typs enthält. Der Response-Typ aus Zeile zehn hat verschiedene Varianten, die angeben, ob die zuvor gesendete Nachricht erfolgreich war oder nicht. Ist die Nachricht erfolgreich verarbeitet worden und die GetCertificate-Variante aus Zeile zwölf wird empfangen, werden

einige Daten mitgeschickt. Im letzten Feld wird die Menge an Bytes geschickt, welche im übergebenen Puffer beschrieben worden sind.

Die beispielhafte Verwendung dieses Traits ist wie in Abbildung 13 dargestellt. Nachrichten werden in Form der Requests der Schnittstelle übergeben. Diese werden serialisiert und werden mithilfe eines Puffers wie dem Ringpuffer aus 6.1 an die CA übergeben. Die CA bearbeitet die Nachricht und schickt in einem zweiten Ringpuffer zurück zur Schnittstelle. Wenn die CE dann die Antworten empfängt, wird die Nachricht aus dem Empfangspuffer genommen. Sie wird in die Form einer Response-Variante überführt und bei Bedarf ein Teil der Daten in übergebenen Puffer geschrieben.

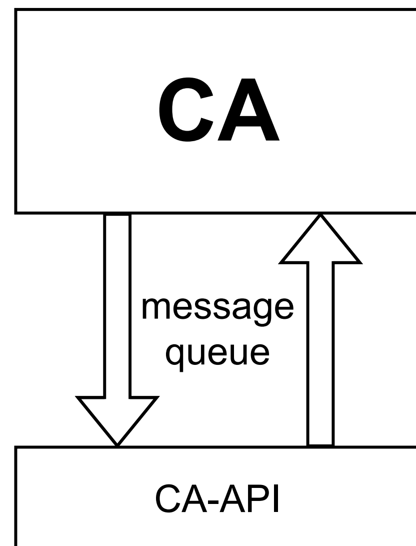

Abbildung 13: Nachrichtenfluss CA

6.2.2 Implementierung interne CA

Die Implementierung des CA-Traits als interne CA für den Rust Demonstrator, ist Gegenstand dieses Kapitel. Auf folgende Aspekte der Implementierung wird näher eingegangen:

- Struktur des internen CA im Detail
- Zertifikatserstellungsprozesses

Während des Entwicklungsprozesses der internen CA wurden verschiedene Bibliotheken getestet, bevor sich wie in Kapitel 4.2 beschrieben für rcgen entschieden wurde. Für diesen Zweck wurden für alle CA-Funktionen ein weiterer Trait geschaffen, der die Austauschbarkeit ermöglicht. Eine Instanz eines solchen CA-Services-Traitobjektes ist Teil der interne CA(siehe Abbildung 14). Weitere Teile sind ein Buffer in dem die Zertifikate von dem CA-Services-Objekt geschrieben werden und ein Ringpuffer.

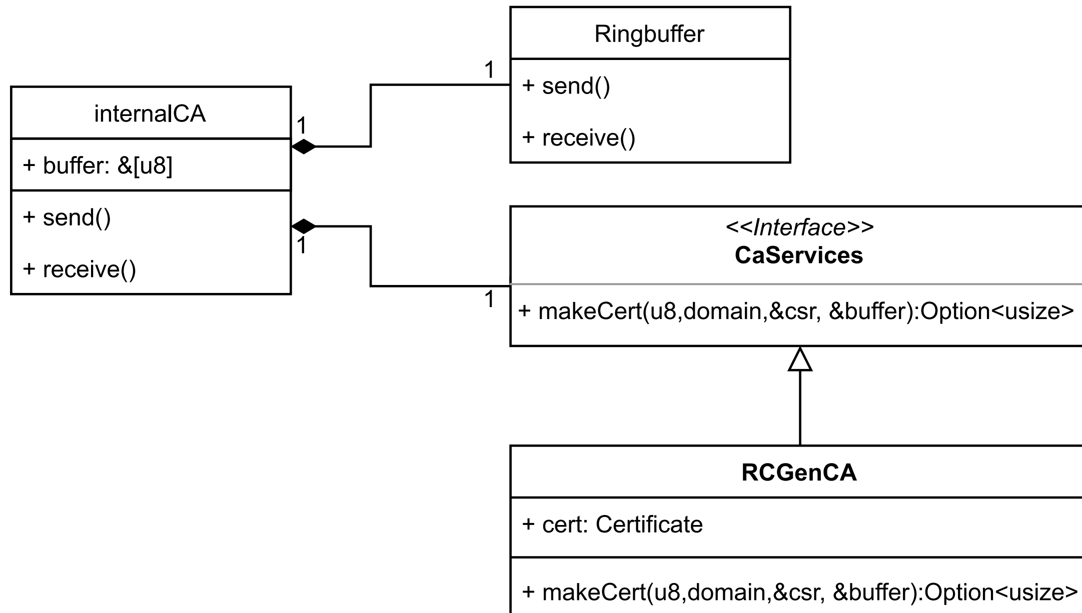

Abbildung 14:Klassendiagramm interne CA

Auffällig ist hier, dass es sich um nur eine Instanz eines Ringpuffers handelt, während der Nachrichtenfluss aus Abbildung 13 für ein CA-Trait-Implementierer zwei Puffer benutzt, um jeweils Sende- und Empfangsrichtung abbilden zu können. Grund für zwei getrennte Nachrichtenpuffer war, dass auch CA-Implementierungen möglich sind, bei welcher die CA auf einem separaten Prozess oder Gerät läuft. Da die interne CA für den Rust-Demonstrator nicht Teil eines separaten Prozesses ist, lässt sich ein Ringpuffer sparen und gleichzeitig die durch den Trait vorgegebene Schnittstelle einhalten.

Hierfür betrachte man den veränderten in Abbildung 15. Der Nachrichtenpuffer für die Senderichtung ist nicht vorhanden, da die Daten direkt der CA übergeben werden können, und diese ihre Arbeit macht, bevor die CE ihre Arbeit fortsetzt. Da die CE jedoch nicht direkt ein fertiges Zertifikat wieder annehmen kann, da dies in der Schnittstelle nicht vorgesehen ist, ist der Puffer für die Rückrichtung notwendig.

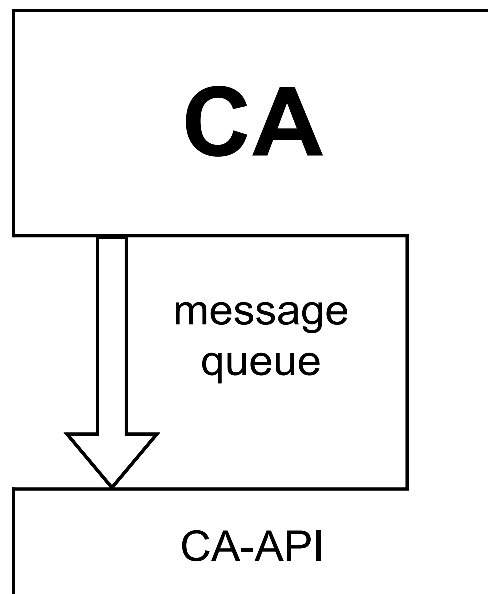

Abbildung 15: Nachrichtenfluss interne CA

Um die Zertifikate für die IA-Komponente in der CA zu erstellen sind 5 Schritte notwendig:

1. Laden des CA-Zertifikats und Schlüssels
2. Einlesen des CSRs
3. Kopieren der Felder für das neue Zertifikat
4. Start- und Enddatum der Gültigkeit setzen
5. Zertifikat erstellen und signieren

6.3 CE

Dieses Kapitel beschäftigt sich mit der Implementierung der Credentialing-Entity, den Details der Struktur und den Herausforderungen die sich beim Implementierungsprozess ergaben. Genauer betrachtet werden hierbei folgende Aspekte:

- CE-Struktur
- Bus-Schnittstelle
- Bewirtschaftungslogik
- C-Schnittstelle

6.3.1 Struktur

Man betrachte die Struktur der Credentialing-Entity in dem untenstehenden Listing 22. Sie besitzt die folgenden Komponenten:

```
1  pub struct CredentialingEntity<'a> {
2      nodes: [Node; NUM_OF_NODES],
3      bus: Can<DtlsBuffer<'a>>,
4      ca: SameThread<rcgen_ca>,
5      mes_buf: [u8; BUFFERSIZE],
6      cred_vault: Vault,
7      ///callbacks
8      is_connected_to_bus: unsafe extern "C" fn(node_id: u8) -> u8,
9      set_dtls_cert: unsafe extern "C" fn(
10         node_id: u8,
11         ca_cert: *const c_char,
12         client_cert: *const c_char,
13         client_key: *const c_char,
14     ) -> bool,
15     get_dtls_state: unsafe extern "C" fn(node_id: u8) -> u8,
16     set_dtls_state: unsafe extern "C" fn(node_id: u8, domain: u8),
17 }
```

Listing 22: Rust-CE Struktur

- nodes

  Eine Liste mit bewirtschafteten Komponenten und deren Status

- bus/ca

  Strukturen für Bus- und CA-Zugriff beschrieben jeweils in 6.3.2 und 6.2

- mes_buf

  Puffer zum empfangen von Bus-/CA-Nachrichten

- cred_vault

  Container für Vertrauensanker die während des Bewirtschaftungsprozesses an IA-Komponenten verschickt werden.

- is_connected_to_bus

  Rückruffunktion um Busverbindungsstatus zu erfragen

- set_dtls_cert

  Rückruffunktion um Zertifikat für DTLS zu bestimmen.

- set/get_dtls_state

  Rückruffunktionen um DTLS-Verbindungsstatus zu ändern/auszulesen

6.3.2 Busschnittstelle

Um Nachrichten mit IA-Komponenten auszutauschen, verwendet die CE einen zweischichtigen Prozess dargestellt in Abbildung 16. Soll eine Nachricht versendet werden, verpackt die CE alle notwendigen Daten in einem Request-Objekt. Dieses wird der ersten Schicht der CAN-Schnittstelle übereicht. Diese Schicht konstruiert die Nachricht in einem Puffer. Dieser Puffer wird dem BusConnector übergeben, welcher die Nachricht an die DTLS-Komponente weitergibt. Umgekehrt werden Nachrichten von der BusConnector-Schicht empfangen und die darunterliegende Schicht decodiert die Nachricht und verpackt sie in einem Response-Objekt.

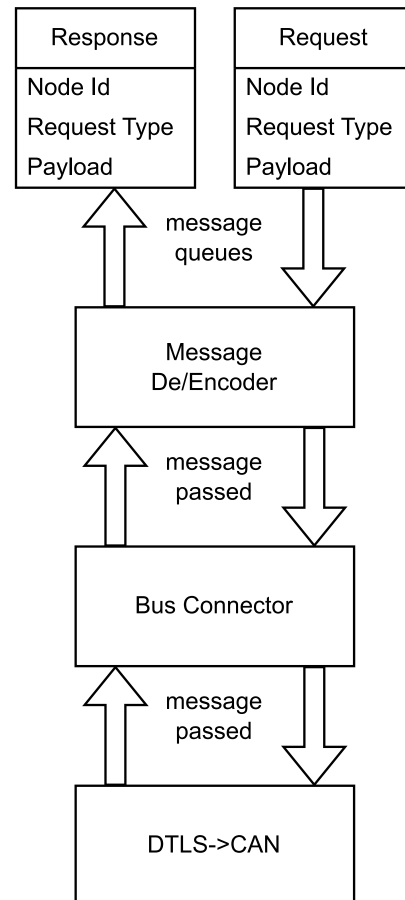

Abbildung 16: CAN-Schnittstelle

Dieses Konzept ist mithilfe einer Struktur implementiert, welche einen austauschbaren BusConnector-Trait verwendet. Die Umsetzung der Request- und Response-Objekte geschieht mithilfe von Rust-enums, welcher für jede Nachrichtenart eine Variante mit den Notwendigen Daten bündelt(siehe Zeile 3 Listing 23). Diese Objekte können der CAN-Struktur übergeben werden. Intern kann dann die entsprechende Variante erkannt werden und die Nachricht korrekt zusammengesetzt werden.

```
1  pub struct Can<T: bus_interface::BusConnector> {
2      buffer: [u8; BUFFERSIZE],
3      connector: T,
4  }
5
6  pub fn send_request(&mut self, req: SouthRequest, node_id: u8) ->
   Option<usize>
7
8  pub enum SouthRequest<'a> {
9      GetCredentialingStatus,
10     GetEECertificationPath(CertificateDomain),
11     TriggerKeyPairGeneration(CertificateDomain,
   KeyPairGenerationAlgorithm),
12     ImprintTrustAnchor(CertificateDomain,
   TrustAnchorCertificate<'a>),
13     ImprintEECertificationPath(CertificateDomain, EeCertPath<'a>),
14     RemoveTrustAnchor(CertificateDomain),
15     RemoveEeCertAndPrivKey(CertificateDomain),
16 }
```

Listing 23: CAN Nachrichten verschicken

Der BusConnector ist mit einem Trait implementiert, welcher je eine Funktion zum Senden und Empfangen von Nachrichten hat. Die Nachrichten werden als serialisierte Byte-Sequenzen in Puffern übertragen.

```
1  pub trait BusConnector {
2          ///tries to send message in passed Buffer
3          /// on Succes returns number of bytes sent
4          fn message_to_bus(&mut self, message: &[u8]) ->
   Option<usize>;
5          ///tries to receive message from bus.
6          ///If message exists writes message into passed message
   buffer and returns number of sent bytes
7          fn message_from_bus(&mut self, message: &mut [u8]) ->
   Option<usize>;
8      }
```

Listing 24: CAN-Struktur

Intern funktioniert der BusConnector, indem zwei Ringpuffer zum Austausch mit der DTLS-Komponente verwendet werden. Diese verwenden jedoch nicht die in 6.1 Rust-Implementierung, sondern eine bereits im Demostrator vorhandene in C++-geschriebene Variante, welche in der DTLS-Komponente integriert sind. Um mit diesem kommunizieren zu können, wurde für den C++-Code Rust-Bindings mit der in 2.2.1 beschriebenen bindgen-Methode erzeugt. Diese Ringpuffer werden dem

DTLS-BusConnector bei dessen Erzeugung als Pointer gegeben und hält diese als Referenzen (siehe Listing 25).

```
1  pub struct DtlsBuffer<'a> {
2      send_buf: &'a mut ringbuffer_t,
3      rec_buf: &'a mut ringbuffer_t,
4  }
```

Listing 25: DTLS-BusConnector

Mithilfe dieser können zugehörige Funktionen aufgerufen werden, um Nachrichten als Byte-Sequenz auszutauschen.

6.3.3 Bewirtschaftungslogik

Für die Bewirtschaftungslogik werden eine Reihe von Zustandsmaschinen verwendet, um die Zustände für jede Komponente einzeln darzustellen. Diese Zustandsmaschinen sind mit Enums implementiert. Sie existieren für den Bus-Verbindungstatus, den DTLS-Verbindungsstatus, ob von der IA-Komponente besessene Zertifikate bereits erfragt worden sind und in welchem Schritt der Bewirtschaftungsprozess ist.

Der erste Schritt der Empfang von Nachrichten vom Bus ist mit while-Schleifen implementiert, welche so lange versuchen je eine Nachricht zu empfangen, bis keine mehr vorhanden sind. Jede empfangene Nachricht wird innerhalb des Schleifenkörpers verarbeitet und bei Bedarf der Bewirtschaftungsstatus einer IA-Komponente verändert.

Im nächsten Schritt werden alle IA-Komponenten in einer for-Schleife der Reihe nach bewirtschaftet. Zunächst wird mit Hilfe der Rückruffunktionen Bus- und DTLS-Status erfragt. Dann wird mithilfe von Match-Statements zunächst die Status von Bus und DTLS ausgewertet. Besteht eine abgesicherte Verbindung, wird mit weiteren Match-Statements der Bewirtschaftungsstatus ausgewertet und die Bewirtschaftung gegebenenfalls mit Bus- oder CA-Nachrichten fortgesetzt.

6.3.4 C-Schnittstelle

Die C-Schnittstelle der CE besteht aus zwei Teilen:

- Rückruffunktionen für DTLS und Bus

- Öffentliche Schnittstelle zum Erzeugen und Abarbeiten der CE

Die Rückruffunktionen sind in C geschriebene Funktionen. Für sie wurde hauptsächlich der Code, der diese Aufgaben in der C++-Implementierung übernahm, angepasst. Die Funktionen werden Rust im Konstruktor der CE-Struktur als Funktionszeiger übergeben. Funktionszeiger werden in Rust wie in Zeile 2 des untenstehenden Listing 26 definiert. In den Klammern stehen Datentypen der Argumente nach dem Pfeil der Datentyp des Rückgabewert.

```
1  //Pointer on Rust-function
2  is_connected: fn(u8) -> u8
3  //Pointer on C-function
4  is_connected: unsafe extern "C" fn(u8) -> u8
```

Listing 26: C-Funktionszeiger in Rust

C-Funktionszeiger benötigen jedoch den Zusatz unsafe extern "C". Extern "C" steht hier für nicht kompatible Binärschnittstellen zwischen C und Rust. Unsafe in Rust bedeutet, dass Rust Teile seiner Sicherheitsgarantien in diesem Code nicht garantieren kann.

Der zweite wichtige Aspekt der C-Schnittstelle, ist der Code zum Erzeugen und Abarbeiten der CE. Hierbei gibt es ein Problem zu lösen. Einerseits sollen interne Details der CE-Struktur nicht von der C-Seite aus gesehen werden, weswegen der Konstruktor nur ein Pointer ausgibt, andererseits kann nun die Struktur nicht auf dem Stack liegen, da die C-Seite nicht weiß wie viel Speicher die Struktur benötigt. Da die Applikation aber auch ohne Heap auskommen soll, muss eine andere Lösung verwendet werden. Ohne Heap und Stack sind globale Variablen eine naheliegende Lösung, welche aber mit einem weiteren Problem kommen. Die Strukturen müssen während der Kompilierung mit einem Literal initialisierbar sein, was aber aufgrund von privaten Feldern in Strukturen der rcgen-Bibliothek nicht möglich ist. Dieses Problem lässt lösen, indem die CE in ein Option-enum gepackt wird, welche mit der None-Variante initialisiert wird (siehe Listing 27).

```
1  static mut CE: Option<CredentialingEntity> = None;
```

Listing 27: CE als globale Variable

Ein weiteres Problem der globalen Struktur ist, dass der Nutzer bei mehrmaligem aufrufen denken könnte, dass es sich um verschiedene Instanzen handelt. Um dies zu verhindern, prüft der Konstruktor, ob die Struktur noch nicht initialisiert wurde und

die None-Variante ist. Ist sie None, wird sie zur Some-Variante mit initialisierten CE und ein Zeiger auf die CE wird zurückgegeben (siehe Zeile 13 Listing 28). Ist die globale Struktur bereits Some, wird ein Null-Zeiger zurückgegeben (Zeile 21).

```
1  if CE.is_none() {
2      CE = Some(
3      CredentialingEntity::new(
4              dtls_sendq,
5              dtls_recq,
6              is_connected,
7              set_dtls_cert,
8              get_dtls_state,
9              set_dtls_state,
10         )
11     );
12     match &mut CE {
13         Some(refr) => return refr as *mut CredentialingEntity,
14         None => {
15         println!("Something went wrong constructing CE-object");
16             ptr::null_mut()
17         },
18     }
19 } else {
20     //Return null on repeated calls
21     ptr::null_mut()
22 }
```

Listing 28: CE-Konstruktor

Der C-Header mit Bindings für den Konstruktor wurde mithilfe von cbindgen, wie in Kapitel 2.2.2 beschrieben, erstellt.

6.4 Integration

Ist die Rust-CE-Bibliothek fertig implementiert, muss sie in die Anwendung integriert werden.

Für die Integration sind drei Schritte notwendig:

1. Inkludieren der Header
2. Konstruktion und Abarbeitung der CE einbauen
3. Build-Skripte für zusätzliche Option anpassen

Da die Rust-Implementierung als austauschbare Alternative zur C-Implementierung dienen soll, muss mit bedingter Kompilierung gearbeitet werden.

```
1  #ifdef USE_RUST_CE
2  run_ce(rustCe);
3  #endif
```

Listing 29: bedingte Kompilierung Rust-CE

Hierfür wird eine #ifdef-Direktive, wie in Listing 29 zu sehen, verwendet. Diese Methode wird für das Inkludieren von Headern, der Initialisierung der CE-Struktur und dem Abarbeiten der CE verwendet.

Als letztes müssen Build-Skripte angepasst werden. Der Demonstrator verwendet CMake und Bash-Skripte, die CMAKE mit passenden Flags aufgerufen und Ordner mit alten Binärdateien säubern. Für das Bash-Skript wird, das Skript für den C++-Build kopiert und die zusätzliche CMAKE-Option RUST_BUILD übergeben. In der CMakeList.txt-Datei wird eine Bedingung (Listing 30) eingebaut, die die Rust-spezifischen Einstellungen wie verwendete Header setzt.

```
1  if(${RUST_BUILD})
2      # Rust specific options
3  endif()
```

Listing 30: CMAKE-Bedingung

7 Validierung

Dieses Kapitel beschäftigt sich mit der Korrektheit der Credentialing-Entity-Implementierung. Hierbei wird auf die Herausforderungen beim Testen dieses Projekts und die verschiedenen durchgeführten Tests eingegangen.

7.1 Herausforderungen beim Testen

Da diese CE-Implementierung eine bereits vorhandene Implementierung ersetzt und eine Test-IA-Anwendung existiert, kann mit der Ersetzung durch die Rust-Variante die vorhandenen Integrationstests übernommen werden.

Hierbei gibt es jedoch ein Problem: Dieser Integrationstest beachtet nur den positiven Pfad, der davon ausgeht, dass alle Nachrichten zwischen CE, CA und IA-Komponente sauber übertragen werden. Da weitere Integrationstests sehr aufwendig in der Implementierung sind, wurde sich entschieden die CA- und IA-Komponenten-Schnittstelle sehr genau mit Komponententests abzudecken.

7.2 Komponententests

Wie im vorigen Kapitel erwähnt, sind Unittests ein zentraler Bestandteil der Teststrategie. Als Testframework wurde das in Rust integrierte cargo-test verwendet.

Folgende Komponenten wurden hierbei getestet:

- MessageDe/Encoder (siehe 6.3.2) und die analog aufgebaute internalCa (siehe 6.2.2)

- CAServices(siehe 6.2.2)

7.2.1 MessageDe/Encoder und internalCa

Diese Schicht hat wie in 6.3.2 beschrieben, zwei Aufgaben. Zum einen werden Nachrichten (De)kodiert zum anderen werden Nachrichten mit dem Bus ausgetauscht. Da der Fokus bei den Tests auf dem Kodieren und Dekodieren der Nachrichten liegt, muss erst ein Ersatz für den BusConnector, der Teil des MessageDe/Encoder ist, erstellt werden. Diese sogenannten Mock-Objekt wird wie

in Abbildung 17 dargestellt mit der Kodierschicht verbunden. Das Mock-Objekt funktioniert folgenderweise. Es besitzt einen internen Puffer, der von außerhalb und durch die Kodierschicht beschrieben und ausgelesen werden kann. Soll ein Sende-Vorgang getestet werden, wird der Puffer nach dem Sendevorgang mit dem Sollinhalt verglichen. Beim Empfangen wird der Puffer vorher mit der empfangenen Nachricht beschrieben und überprüft, ob das passende Response-Objekt erzeugt wird. Auch lässt sich einstellen, ob Sende- und Empfangsvorgänge erfolgreich sind

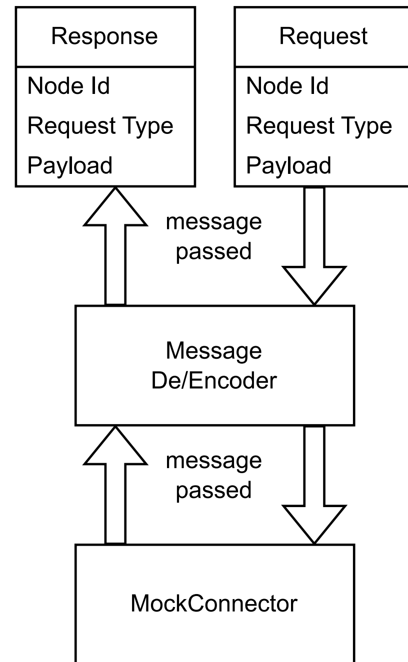

Abbildung 17: Mock-Schnittstelle

Bei dieser Komponente werden folgende Testfälle beachtet:

Senden von

- Verschiedenen korrekt kodierten Nachrichten

Inkorrekte Request-Objekte sind unmöglich durch Verwendung von Enums.

Empfang von

- verschiedenen korrekten Nachrichten
- Nachrichten mit korrekter ID aber zu ungültigem Inhalt/ Inhalt falscher Länge
- Nachrichten mit ungültiger ID
- zu kurzen Nachrichten, sodass Nachrichten-ID oder Inhalt fehlen.

Zusätzlich wird überprüft, ob abgelehnte Sendungen und Empfänge korrekt weitergegeben werden.

Für die interne CA werden analog ein Mock-Objekt des CAServices-Traits erstellt, und Testfälle erstellt und durchgeführt.

7.2.2 CAServices

Da bei den Mock-Tests der CA-Schnittstelle nicht mit der richtig funktionierenden CA gearbeitet wurde, muss überprüft werden, ob die erstellten Zertifikate gültig sind. Hierfür wurde ein CSR von der IA-Komponente aus dem C++-Demonstrator abgefangen und gespeichert. Mit diesem wurde folgender Testablauf durchgeführt:

1. CSR der CA geben
2. Überprüfen, ob Zertifikatserstellung erfolgreich ist
3. Zertifikat mit anderer Zertifikatsbibliothek (hier Openssl) einlesen
4. Überprüfen, ob das Einlesen erfolgreich ist

7.3 Integrationstests

Ist die Rust-CE in die CE-Anwendung integriert, muss sie gegen die IA-Testkomponente getestet werden. Hierbei wir folgendermaßen vorgegangen.

1. C++-CE und IA bauen, den Ablauf einer fehlerlosen Bewirtschaftung beobachten
2. C++-CE gegen Rust Variante austauschen
3. Rust-CE testen und beobachten
4. Bei unerwartetem Verhalten Fehler suchen und ausbessern und zurück zu 3.

Durch das Testen mit dieser Methode konnten 2 Fehler erfolgreich gefunden und behoben werden.

- Rcgen erkennt von Openssl erzeugte Private Schlüssel nicht

  Durch Recherche konnte herausgefunden werden, dass Openssl standardmäßig private Schlüssel im Format von #PKCS8v1 erzeugt. Die von rcgen verwendete Crypto-Bibliothek namens ring unterstützt jedoch nur das neuere #PKCS8v2-Format. Als Lösung wurden Ersatzzertifikate mit Schlüsseln von rcgen erzeugt [40].

- CSRs die von der IA-Komponente kommen, können nicht gelesen werden

Die Ursache für diesen Defekt konnte ein Fehler in der Protokoll-Implementierung für diese Nachricht ausgemacht werden. Bei der Implementierung wurde mit einer veralteten Spezifikation gearbeitet. Dieser fehlte, dass dem CSR ein weiteres Byte mit zusätzlichen Informationen vorangestellt wird, welches in der fehlerhaften Version als Teil des CSR interpretiert wurde und somit den CSR korrumpierte. Zur Lösung wurde das Protokoll angepasst.

Nach Anpassung dieser beiden Aspekte, funktionierte die Rust-CE wie gewünscht.

8 Zusammenfassung

Im Rahmen dieser Arbeit wurde das Modell der Credentialing-Entity in der Programmiersprache Rust implementiert. Hierbei wurde die Nutzung von Rust als Ergänzung oder Alternative zu C und C++ in mehreren Aspekten beleuchtet.

Die Sprachfeatures die Rust Speichersicherheit ohne Performanceeinbußen einbringen, wurden theoretisch beleuchtet.

In der Implementierung wurden praktische Aspekte der Verwendung von Rust betrachtet:

- Verfügbare Bibliotheken in den Bereichen (D)TLS/X509-Zertifikaten:

  Rust ist im Vergleich zu C und C++ eine sehr junge Programmiersprache. Dies spiegelt sich auch im Rust-Ökosystem wider. Eine allumfassende TLS- und Zertifikatsbibliothek wie OpenSSL oder mbedTLS gibt es zum Zeitpunkt dieser Arbeit keine. Zertifikatsbibliotheken gibt es viele, jedoch fehlen allen Features, sodass mehrere verwendet werden müssen. Im Allgemeinen muss bei der Verwendung von Rust im Voraus recherchiert werden, ob für verwendete C-Bibliotheken, gute Rust-Alternativen existieren.

- C/C++-Interoperabilität:

  Sowohl für den Zweck des Schreibens einer C/C++ Bibliothek in Rust als auch das Schließen der Lücken im Rust-Ökosystem mit C/C++-Bibliotheken, wurde die Interoperabilität zwischen Rust und C/C++ untersucht. Hierbei konnte festgestellt werden, dass dank einiger sehr guter Tools, beide Sprachen recht einfach zusammen verwendet werden, was den potentiellen Mehraufwand gering hält.

- Portierung und Sprachunterschiede

  Softwarearchitekturen in Rust können sich drastisch von C/C++-Architekturen unterscheiden. Grund hierfür können zum einen Schwierigkeiten durch fehlende Sprachkonzepte wie Vererbung sein oder Rust hat zusätzliche Sprachfeatures die die Art der Codestrukturierung stark beeinflussen. Der Rust-Borrowchecker

und die dahinterstehenden Regeln können zusätzlich erschwerend wirken. Aber Rust bietet auch Vorteile. Der Borrowchecker und die Regeln dahinter bieten Schutz vor gängigen Fehlern beim Speichermanagement, welche in der Vergangenheit für viele kritische Schwachstellen sorgten. Speziell der Schutz vor Pufferüberläufen ist sehr wichtig. Ein weiter Vorzug von Rust sind Rust-Enums, mithilfe derer sich sehr einfach Zustandsmaschinen und Strukturen ohne ungültige Zustände modellieren lassen.

Abschließend lässt sich sagen, dass sich die Verwendung von Rust als Ergänzung zu vorhandenem C/C++-Code lohnen kann. Vor allem wenn mit nicht vertrauenswürdigen Daten gearbeitet wird, können die Schutzmechanismen in Rusts Speichermanagement eine wertvolle Ergänzung sein. So sind Pufferüberläufe, die für mehrere kritische Schwachstellen in bekannten TLS-Bibliotheken verantwortlich sind [41, 42], eine der Fehlerquellen vor denen Rust schützt. Im Einzelfall muss jedoch darauf geachtet werden, ob das Rust-Ökosystem notwendige Bibliotheken liefert und ob der Mehrwert der Sprache die Nachteile einer gemischten Codebase rechtfertigen.

9 Abkürzungsverzeichnis

9.1 Abkürzungen

CE *Credentialing Entity*
CRL *Certificate-Revocation-List*
CSR *Certificate Signing Request*, *Certificate Signing Request*
DTLS *Datagram Transport Layer Security*
IA *Industrieautomatisierung*
OT *Operational Technologies*
PKI *Public Key Infrastructure*
RPKI *Resource-Public-Key-Infrastructure*

9.2 Begriffe und Definitionen

Public-Key-Infrastructure: „Eine Public Key Infrastruktur (PKI, Infrastruktur für öffentliche Schlüssel) ist ein hierarchisches System zur Ausstellung, Verteilung und Prüfung von digitalen Zertifikaten.“ [43]

X509-Zertifikate: X509-Zertifikate sind digitale Zertifikate. „[…] digitalen Zertifikate ermöglichen eine vertrauenswürdige Zuordnung von Entitäten zu ihren öffentlichen Schlüsseln.“ [43]

Abbildungsverzeichnis

Abbildung 1: Rust-Enum 17
Abbildung 2: CAN-Demonstrator Grobstruktur 18
Abbildung 3: C++-Strukturdiagramm 19
Abbildung 4: Klassendiagramm CE 20
Abbildung 5: Hauptschleife C++-Programm 22
Abbildung 6: Flussdiagramm Grobablauf CE 23
Abbildung 7: Flussdiagramm Verbindungsaufbau 24
Abbildung 8: Sequenzdiagramm: Bewirtschaftung 25
Abbildung 9: Grobarchitektur Gesamtsystem Rust 36
Abbildung 10: Klassendiagramm Rust-CE 37
Abbildung 11: Funktionsprinzip Ringpuffer 40
Abbildung 12: Aufbau Ringpuffer-Nachrichten 41
Abbildung 13: Nachrichtenfluss CA 43
Abbildung 14:Klassendiagramm interne CA 44
Abbildung 15: Nachrichtenfluss interne CA 44
Abbildung 16: CAN-Schnittstelle 47
Abbildung 17: Mock-Schnittstelle 54

10 Anhang

10.1 Untersuchung TLS

Anforderung	rustls	Embedded-TLS	WebRTC-DTLS	SaiTLS
TLS 1.2/1.3	Ja	Nur TLS 1.3 nur Aes128GcmSha256 Ciphersuite	Nur DTLS 1.2	Nur TLS 1.2
DTLS	Nein	Nein	Ja	Nein
TLS CRL-Support	Nein	Nein	Nein	Nein
Lesen und schreiben von X.509-Zertifikaten	Nur lesen für interne Validierung	Nur lesen für interne Validierung	Nur lesen für interne Validierung	Nur lesen für interne Validierung
Lesen von CRLs	Nein	Nein	Nein	Nein
Lesen und schreiben von CSRs	Nein	Nein	Nein	Nein
No_STD	Nein	Ja	Nein	Nein

10.2 Untersuchung Zertifikatsbibliotheken

Anforderung	rpki	X509-cert	Barebones-x509	x509-certificate	picky	X.509 Parser	rcgen
Zertifikate validieren	Ja	Nein	Keine Zertifikatsketten	Ja	Ja	ja	basiert auf X.509 Parser[44]
X.509-Zertifikate DER und PEM	Nur im Kontext von RPKIs[45]	Ja	Nein	Ja	Ja	Ja	Ja
Zertifikatsfelder lesen	Ja	Ja	Eingeschränkt	Ja	Ja	Ja	Ja
Lesen von CRLs	Nein	Ja Extension nicht decoded	Nein	Ja	Nein	Ja	Ja
Lesen/schreiben von CSRs	Nur im Kontext von RPKIs[45]	Ja	Nein	Ja	Ja	Ja	Ja
Erstellen von Zertifikaten	Nein	Nein	Nein	Ja	Ja, ECDSA fehlt	Nein	Ja
No_STD	Nein	Ja, aber eingeschränkt	Ja, aber eingeschränkt	Ja	Nein	Nein	Nein
Aktivität im Repository	Mehrere aktiv Mitwirkende	Mehrere aktiv Mitwirkende	Einzelprojekt, Keine Aktivität	Mehrere aktiv Mitwirkende	Mehrere aktiv Mitwirkende	mehrere Mitwirkende	mehrere aktiv Mitwirkende
Lizenz	BSD-3	MIT oder Apache-2.0	MIT oder Apache-2.0	MPL-2.0	MIT oder Apache-2.0	MIT oder Apache-2.0	MIT oder Apache-2.0

11 Literatur

[1] Agentur triebwerk GmbH, Nuernberg, www.agentur-triebwerk.de. „IT-Sicherheit: Secure Fieldbus Communication.” https://infoteam.de/pressebereich/pressemitteilungen/detail/it-sicherheit-secure-fieldbus-communication (Zugriff am: 27. Februar 2023).

[2] „SD_atp_magazin_05_2022_hb_niemann_V02,” [Online]. Verfügbar unter: https://serwiss.bib.hs-hannover.de/frontdoor/deliver/index/docId/2320/file/SD_atp_magazin_05_2022_hb_niemann_V02.pdf

[3] „SD_atp_magazin_05_2022_hb_niemann_V02,” [Online]. Verfügbar unter: https://serwiss.bib.hs-hannover.de/frontdoor/deliver/index/docId/2320/file/SD_atp_magazin_05_2022_hb_niemann_V02.pdf

[4] „Institut für verlässliche Embedded Systems und Kommunikationselektronik.” https://ivesk.hs-offenburg.de/ (Zugriff am: 27. Februar 2023).

[5] „Deutsche Forschungsgesellschaft für Automatisierung und Mikroelektronik e.V.” https://www.dfam.de/ (Zugriff am: 27. Februar 2023).

[6] Amazon Web Services. „Sustainability with Rust | Amazon Web Services.” https://aws.amazon.com/de/blogs/opensource/sustainability-with-rust/?tag=zd-buy-button-20&ascsubtag=1ddc6af3e945494ebd3b89e022b5e7ac%7C7506ddb7-b5b4-4382-96d3-4199550d5c29%7Cdtp-oo (Zugriff am: 24. Februar 2023).

[7] Wikipedia. „Rust (Programmiersprache).” https://de.wikipedia.org/w/index.php?title=Rust_(Programmiersprache)&oldid=227234267 (Zugriff am: 27. Oktober 2022).

[8] „Comparing Performance: Loops vs. Iterators - The Rust Programming Language.” https://doc.rust-lang.org/book/ch13-04-performance.html (Zugriff am: 27. Oktober 2022).

[9] „A proactive approach to more secure code – Microsoft Security Response Center.” https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/ (Zugriff am: 28. Oktober 2022).

[10] „What is Ownership? - The Rust Programming Language.” https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html (Zugriff am: 28. Oktober 2022).

[11] „Validating References with Lifetimes - The Rust Programming Language." https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html (Zugriff am: 18. Januar 2023).
[12] „Build Scripts - The Cargo Book." https://doc.rust-lang.org/cargo/reference/build-scripts.html (Zugriff am: 20. Oktober 2022).
[13] „Introduction - The `bindgen` User Guide." https://rust-lang.github.io/rust-bindgen/ (Zugriff am: 20. Oktober 2022).
[14] „Generating Bindings to C++ - The `bindgen` User Guide." https://rust-lang.github.io/rust-bindgen/cpp.html (Zugriff am: 20. Oktober 2022).
[15] GitHub. „GitHub - rust-lang/rust-bindgen: Automatically generates Rust FFI bindings to C (and some C++) libraries." https://github.com/rust-lang/rust-bindgen (Zugriff am: 20. Oktober 2022).
[16] „Library Usage with build.rs - The `bindgen` User Guide." https://rust-lang.github.io/rust-bindgen/library-usage.html (Zugriff am: 20. Oktober 2022).
[17] „FFI - The Rustonomicon." https://doc.rust-lang.org/nomicon/ffi.html (Zugriff am: 21. Oktober 2022).
[18] „Other reprs - The Rustonomicon." https://doc.rust-lang.org/nomicon/other-reprs.html (Zugriff am: 21. Oktober 2022).
[19] „cbindgen - crates.io: Rust Package Registry." https://crates.io/crates/cbindgen (Zugriff am: 6. Januar 2023).
[20] GitHub. „cbindgen/docs.md at b6e73017e679caf3b01217e62642d0722a464887 · eqrion/cbindgen." https://github.com/eqrion/cbindgen/blob/HEAD/docs.md (Zugriff am: 7. Januar 2023).
[21] GitHub. „cbindgen/docs.md at b6e73017e679caf3b01217e62642d0722a464887 · eqrion/cbindgen: cbindgen.toml." https://github.com/eqrion/cbindgen/blob/HEAD/docs.md #cbindgentoml (Zugriff am: 7. Januar 2023).
[22] Stack Overflow. „Stack Overflow Developer Survey 2022." https://survey.stackoverflow.co/2022/ (Zugriff am: 22. Februar 2023).
[23] „Copyleft. Was ist das? - GNU-Projekt - Free Software Foundation." https://www.gnu.org/licenses/copyleft.de.html (Zugriff am: 20. September 2022).
[24] „Open Source Licensing Guide | New Media Rights." https://www.newmediarights.org/open_source/new_media_rights_open_source_licensing_guide (Zugriff am: 20. September 2022).

[25] Mend. „Open Source Licenses in 2022: Trends and Predictions.” https://www.mend.io/resources/blog/open-source-licenses-trends-and-predictions/ (Zugriff am: 20. September 2022).
[26] „crates.io: Rust Package Registry.” https://crates.io/ (Zugriff am: 20. September 2022).
[27] GitHub. „GitHub: Where the world builds software.” https://github.com/ (Zugriff am: 20. September 2022).
[28] GitHub. „GitHub - rustls/rustls: A modern TLS library in Rust.” https://github.com/rustls/rustls (Zugriff am: 19. September 2022).
[29] GitHub. „GitHub - drogue-iot/embedded-tls: An Rust TLS 1.3 implementation for embedded devices.” https://github.com/drogue-iot/embedded-tls (Zugriff am: 19. September 2022).
[30] GitHub. „GitHub - webrtc-rs/webrtc: A pure Rust implementation of WebRTC.” https://github.com/webrtc-rs/webrtc/ (Zugriff am: 19. September 2022).
[31] Renet. „SaiTLS.“ (Zugriff am: 19. September 2022).
[32] GitHub. „GitHub - NLnetLabs/rpki-rs: An RPKI library for Rust.“ (Zugriff am: 19. September 2022).
[33] GitHub. „GitHub - NLnetLabs/rpki-rs: An RPKI library for Rust.“ (Zugriff am: 6. Februar 2023).
[34] „x509-certificate - crates.io: Rust Package Registry.” https://crates.io/crates/x509-certificate (Zugriff am: 21. September 2022).
[35] GitHub. „GitHub - barebones-x509/barebones-x509: Low-level X.509 verification.” https://github.com/barebones-x509/barebones-x509 (Zugriff am: 19. September 2022).
[36] GitHub. „GitHub - Devolutions/picky-rs: Picky portable PKI implementation and microservice.” https://github.com/Devolutions/picky-rs (Zugriff am: 19. September 2022).
[37] GitHub. „GitHub - rusticata/x509-parser: X.509 parser written in pure Rust. Fast, zero-copy, safe.” https://github.com/rusticata/x509-parser (Zugriff am: 19. September 2022).
[38] „rcgen - crates.io: Rust Package Registry.” https://crates.io/crates/rcgen (Zugriff am: 5. Januar 2023).

[39] „Object Oriented Programming Features of Rust - The Rust Programming Language.” https://doc.rust-lang.org/book/ch17-00-oop.html (Zugriff am: 19. Februar 2023).

[40] GitHub. „Problem creating self signed cert with ECDSA algorithm and using as client identity in native-tls · Issue #82 · est31/rcgen.” https://github.com/est31/rcgen/issues/82 (Zugriff am: 22. Februar 2023).

[41] F. E. Team, „OpenSSL Vulnerability 2022: Details and Fixes - FOSSA,“ *Dependency Heaven*, 31. Oktober 2022. https://fossa.com/blog/openssl-vulnerability-2022-details-fixes/ (Zugriff am: 27. Februar 2023).

[42] S. Luber, „Was ist Heartbleed?,“ *Security-Insider*, 16. November 2022. https://www.security-insider.de/was-ist-heartbleed-a-899d5093e814fd8f5b709a5b2166395f/ (Zugriff am: 27. Februar 2023).

[43] Bundesamt für Sicherheit in der Informationstechnik. „Public Key Infrastrukturen - Public Key Infrastrukturen (PKIen).” https://www.bsi.bund.de/DE/Themen/Oeffentliche-Verwaltung/Elektronische-Identitaeten/Public-Key-Infrastrukturen/public-key-infrastrukturen.html (Zugriff am: 27. Februar 2023).

[44] „rcgen - Rust.” https://docs.rs/rcgen/0.10.0/rcgen/index.html (Zugriff am: 5. Januar 2023).

[45] GitHub. „GitHub - NLnetLabs/rpki-rs: An RPKI library for Rust.” https://github.com/NLnetLabs/rpki-rs/issues/205 (Zugriff am: 21. September 2022).