Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Sascha Klaholz

Portierung der OpenSSL Bibliothek auf Knoten eines Wireless Sensor Networks mit dem Betriebssystem RIOT

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Sascha Klaholz

Portierung der OpenSSL Bibliothek auf Knoten eines Wireless Sensor Networks mit dem Betriebssystem RIOT

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thomas C. Schmidt
Zweitgutachter: Prof. Dr. Wolfgang Fohl

Eingereicht am: 02. Dezember 2013

Sascha Klaholz

Thema der Arbeit

Portierung der OpenSSL Bibliothek auf Knoten eines Wireless Sensor Networks mit dem Betriebssystem RIOT

Stichworte

Portierung, OpenSSL, MatrixSSL, eingebettetes SSL, RIOT-OS, MSB-A2

Kurzzusammenfassung

Die Anwendungsgebiete für Wireless Sensor Networks wachsen stetig. Mit den neuen Gebieten wächst auch der Bedarf nach Möglichkeiten, die Kommunikation unter den Knoten abzusichern. Die Absicherung eines Bereiches unter Verwendung eines Wireless Sensor Netzwerks zum Beispiel bedarf einer sicheren Kommunikation der Knoten. Da das SSL-Protokoll etablierter Standard für sichere Kommunikation ist, beschäftigt sich diese Bachelor Arbeit mit der Portierung des SSL Protokolls auf das RIOT Betriebssystem. Im Rahmen dieser Bachelorarbeit wurde festgestellt, dass die OpenSSL Bibliothek für eine Portierung ungeeignet ist. Alternative SSL Implementierungen für eingebettete Systeme werden evaluiert und die MatrixSSL Implementierung als Lösung ausgewählt. Diese konnte erfolgreich portiert und getestet werden.

Sascha Klaholz

Title of the paper

Porting of the OpenSSL library to nodes of a wireless sensor network using the RIOT operating system

Keywords

Porting, OpenSSL, MatrixSSL, embedded SSL, RIOT-OS, MSB-A2

Abstract

The field of application for wireless sensor networks is growing steadily. With these new fields of application there is an increased need for opportunities to secure the communication between nodes. Area monitoring using wireless sensor networks for example require a secure communication among the nodes. Because SSL is the established standard for secure communication, this bachelor thesis deals with porting the SSL-protocol to the RIOT operating system. As part of this work it became apparent that OpenSSL library is unsuitable for porting. Alternative implementations of SSL for embedded systems are evaluated and MatrixSSL is chosen as solution. It was successfully ported and tested.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Zur Motivation dieser Arbeit

Ein Wireless Sensor Network (WSN) besteht aus vielen einzelnen Geräten, den sogenannten Knoten. Ein Knoten ist ein Gerät, welches aus einem Prozessor, Speicher, einem oder mehreren Sensoren sowie einer Stromversorgung besteht. Zur Kommunikation zwischen den Knoten innerhalb des Netzwerks wird ein drahtloses Interface benutzt. Die Anwendungsgebiete von WSNs sind dabei vielfältig. Sie können zur Überwachung von großen Gebieten eingesetzt werden. Zum Beispiel in einem besonders brandgefährdeten Abschnitt eines Waldes. Hier werden die vorhandenen Sensoren der Knoten benutzt, um regelmäßig die Temperatur zu messen. Bei kritischen Werten besteht die Möglichkeit frühzeitig zu reagieren. Auch finden WSN Anwendung in der Überwachung von sicherheitskritischen Bereichen, wo sie bei Aufzeichnungen von verdächtigen Aktionen eine Meldung auslösen können.
Eine Hauptanforderung an die Knoten eines WSN ist der möglichst geringe Energiebedarf, da sie, bedingt durch ihr Einsatzgebiet, in der Regel ohne eine externe Energieversorgung auskommen müssen. Sie sind auf einen Batteriebetrieb über eine lange Zeitspanne hinweg angewiesen. Wegen des geringen Energieverbrauchs ist 6LoWPAN Montenegro u. a. (2007) als Protokoll für die drahtlose Verbindungen in diesem Umfeld sehr gut geeignet. Es wurde von Grund auf für Anwendungen entwickelt, die eine hohe Energieeffizienz bieten sollen. Weiterhin ist die Energierestriktion nur mit Hardwarekomponenten zu erfüllen, die zwar in ihrer Leistungsfähigkeit stark beschränkt sind, dafür eine sehr geringe Leistungsaufnahme haben. Ein Einsatz von klassischen Betriebssystemen ist auf dieser Hardware nicht möglich und die bereits für WSNs vorhandenen Systeme sind in ihren Möglichkeiten sehr eingeschränkt. Das RIOT Betriebssystem hat sich das Ziel gesetzt, auch auf sehr leistungsschwacher Hardware die Vorzüge eines klassischen Betriebssystems nutzbar zu machen.
Durch die in der Regel großflächigen, öffentlichen Einsatzgebiete, sowie dem Einsatz von drahtloser Kommunikation, sind die Knoten vor dem potentiellen Zugriff von

unautorisierten Personen ungeschützt. Dabei gibt es viele Motivationen für einen solchen Angriff. Ein Angreifer könnte sich durch die Manipulation der übertragenen Daten eines Knoten Zugang zu einer Sicherheitseinrichtung verschaffen. Aber auch eine rein destruktive Absicht könnte eine unautorisierte Person veranlassen, die Daten eines WSNs zu manipulieren. Dabei besteht die Gefahr, dass die über lange Zeit gesammelten Aufzeichnungen durch die Datenmanipulation zunichte gemacht werden. Auch der reine Inhalt von ausgetauschten Daten kann für Außenstehende von Interesse sein. Besonders wenn diese aus einem sensiblen Bereich stammen. Denkbar ist hier, dass durch das installieren von Geräten innerhalb des WSNs, deren ungeschützter Datenaustausch über die drahtlosen Interfaces dauerhaft aufgezeichnet wird. Der Einsatz von Verschlüsselung ist ein effektiver Schutz gegen diese Angriffsszenarien.

1.2 Zur Aufgabenstellung dieser Arbeit

Ziel dieser Arbeit ist es, den Einsatz von Verschlüsselung auf dem RIOT Betriebssystem zu ermöglichen. Ein etablierter Standard für verschlüsselte Kommunikation ist der Secure Sockets Layer (SSL). In dieser Arbeit soll die freie SSL Implementierung OpenSSL auf RIOT portiert werden.

1.3 Der Aufbau der Arbeit

Das erste Kapitel beschreibt die Motivation und die Aufgabenstellung dieser Arbeit.
Das zweite Kapitel vermittelt die Grundlagen, die für das Verständnis dieser Arbeit nötig sind. Dazu zählen ein kurzer Überblick über Begriffe im Umfeld der Kryptographie, und die Beschreibung der Funktionsweise von SSL.
Im dritten Kapitel wird analysiert, welche Schritte notwendig sind, um OpenSSL zu portieren. Dazu werden die Kernkomponenten von OpenSSL vorgestellt und es werden Ansätze aufgezeigt, um die Bibliothek auf die Zielplattform zu portieren.
Das vierte Kapitel beschreibt die eigentliche Portierung.
Im fünften Kapitel werden Tests durchgeführt und validiert, ob die im Rahmen dieser Arbeit erfolgte Portierung korrekt umgesetzt worden ist.
Kapitel sechs fasst die während der Arbeit gewonnenen Erkenntnisse zusammen und gibt einen Ausblick auf die Zukunft.

2 Grundlagen

Dieses Kapitel gibt einen Überblick über die grundlegenden Begriffe im Bereich der Kryptographie. Es werden Verfahren beschrieben, die von SSL zur Sicherung der Verbindung eingesetzt werden. Weiterhin wird die Funktionsweise von SSL anhand der einzelnen Schritte beim Aufbau einer Verbindung dargestellt.

2.1 Kryptographiefunktionen

2.1.1 Symmetrische Verschlüsselung

Bei symmetrischen Verfahren wird derselbe Schlüssel sowohl zum Ver- als auch Entschlüsseln verwendet. Damit braucht jeder Kommunikationspartner nur genau diesen Schlüssel. Bedingt durch ihre mathematische Grundlage sind diese Verfahren in der Regel um ein Vielfaches schneller als ihre asymmetrischen Pendants. Allerdings besteht das sogenannte Schlüsselverteilungsproblem, denn jedem Partner muss der Schlüssel zur Verfügung gestellt werden. Es reicht aus, dass bei einem der Kommunikationspartner der Schlüssel kompromittiert wird, um einem Angreifer die Entschlüsselung der gesamten Kommunikation zu ermöglichen. Verbreitete Verfahren sind AES (Advanced Encryption Standard) NIST197 (2001) und DES (Data Encryption Standard) Des (1977).

2.1.2 Asymmetrische Verschlüsselung

Asymmetrische Verschlüsselungsverfahren nutzen unterschiedliche Schlüssel zum Chiffrieren und Dechiffrieren. Jeder Kommunikationspartner generiert bei diesem Verfahren ein eng verknüpftes Schlüsselpaar. Es besteht aus einem öffentlichen und einem geheimen Schlüssel. Der öffentliche Schlüssel wird jedem Kommunikationspartner zur Verfügung gestellt, der geheime bleibt bei jedem Partner und darf niemand anderem als ihm bekannt sein. Eine mit dem ersten Schlüssel des Paares verschlüsselte Nachricht kann ausschließlich mit dem zweiten entschlüsselt werden. Das bekannteste Verfahren ist RSA Rivest u. a. (1978).

2.1.3 Hybride Verschlüsselung

Die hybride Verschlüsselung verfolgt den Ansatz, die beschriebenen Nachteile von symmetrischer als auch asymmetrischer Verschlüsselung zu eliminieren. Dazu werden beide Verfahren kombiniert, die Chiffrierung der zu übertragenen Nutzdaten wird symmetrisch durchgeführt, der Austausch des dazu nötigen Schlüssels erfolgt asymmetrisch. So wird das Schlüsselaustauschproblem umgangen, während gleichzeitig die hohe Geschwindigkeit von symmetrischer Verschlüsselung genutzt werden kann. SSL benutzt hybride Verschlüsselung, um diese Vorteile auszunutzen.

2.1.4 Hashfunktion

Eine kryptographische Hashfunktion ist eine Funktion, die aus sehr großen Eingaben kleine Ausgaben erzeugt, den sogenannten Hashwert. Dieser Hashwert ist vergleichbar mit einem „Fingerabdruck“ einer Nachricht oder eines Dokuments, er lässt keine Rückschlüsse auf die Nachricht zu. Mit dieser Funktion ist es praktisch unmöglich zwei gleiche Hashwerte zu zwei unterschiedlichen Nachrichten zu erzeugen. Sollte dies gelingen, spricht man von einer Kollision. Verbreitete Hashfunktionen sind MD5 Rivest (1992) und SHA-1 Eastlake und Jones (2001).

2.1.5 Digitale Signaturen

Durch den Einsatz von digitalen Signaturen kann ein Empfänger verifizieren, ob eine Nachricht von dem Sender stammt, für den er sich ausgibt. Ermöglicht wird diese Verifikation durch Verwendung von asymmetrischer Verschlüsselung. Der Sender signiert die zu versendende Nachricht mit seinem privaten Schlüssel. Dabei ist die Signierung nicht für die gesamte Nachricht nötig, es reicht aus einen Hashwert über die Nachricht zu bilden und diesen zu signieren. Der Nachteil der langsamen Geschwindigkeit asymmetrischer Verfahren kann so vernachlässigt werden, weil nur mit einer sehr kleinen Datenmenge, dem Hashwert, gearbeitet wird. Jeder Empfänger der im Besitz des öffentlichen Schlüssels des Senders ist, kann mit diesem überprüfen, ob die Nachricht wirklich von dem Sender unterzeichnet worden ist.

2.2 SSL

Die erste Spezifikation von TCP ist Mitte der 70iger Jahre erschienen, die Standardisierung erfolgte 1981 Postel (1981). Ziel war es, ein Protokoll zu schaffen, mit dem Daten

zuverlässig über das Internet transportiert werden konnten. Zu diesem Zeitpunkt stand die Sicherheit dieser Daten noch nicht im Fokus, deshalb sieht TCP keinerlei Mechanismen zur Sicherstellung von Integrität, Vertraulichkeit oder Authentizität vor. Erst etwa 20 Jahre später wurde von Netscape die erste Version von Secure Socket Layer (SSL) veröffentlicht. Es setzt dabei auf der Transportschicht auf, mit dem Ziel Funktionen zum sichern der zu übertragenen Daten zu gewährleisten. Anfang 1999 wurde das Protokoll von der IETF mit der RFC2246 Dierks und Allen (1999) als Standard festgelegt und in Transport Layer Security (TLS) umbenannt. Seitdem wurde das Protokoll um viele Funktionen erweitert, und liegt mittlerweile in der Version 1.3 vor. TLS selbst setzt sich aus zwei Protokollen zusammen, dem TLS Record- und dem TLS Handshake Protokoll. Das Record Protokoll bildet dabei die untere Schicht von beiden. Zu seinen Aufgaben gehört die Komprimierung, Fragmentierung und die eigentliche Übertragung der Daten. Direkt darauf setzt das Handshake Protokoll auf, es stellt die nötigen Funktionen zum Aushandeln der Sicherheitsparameter und der Authentizitätsprüfung zur Verfügung.

2.2.1 Handshake Protokoll

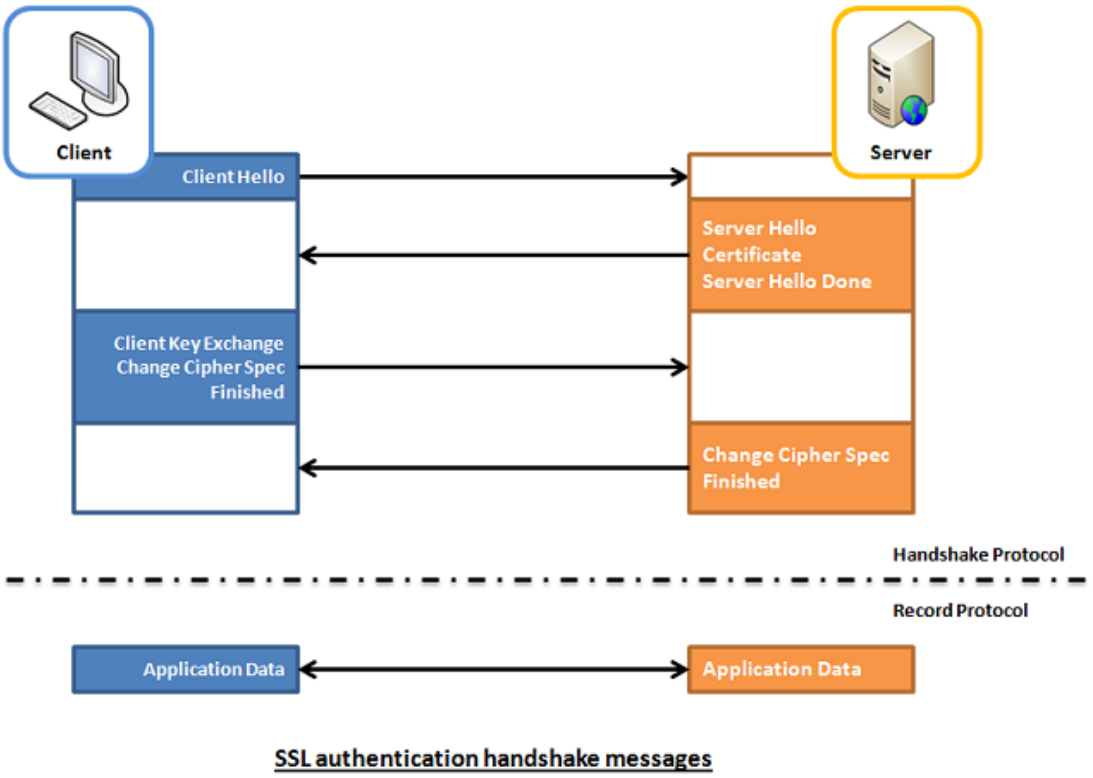

Abb. 2.1: SSL Handshake

1. Der Client initiiert den Aufbau einer gesicherten Verbindung mit einem Server. Er sendet eine *ClientHello* Nachricht. Diese enthält eine generierte Zufallszahl mit Zeitstempel, Protokollversion, Kompressionsalgorithmen sowie die verfügbaren Cipher Suites.
2. Die Nachricht des Clients wird vom Server mit *ServerHello* beantwortet. Darin befindet sich die von beiden Seiten unterstützte höchste Protokollversion, die ID der Sitzung, eine Zufallszahl mit Zeitstempel und Kompressionsalgorithmen. Außerdem wählt der Server die stärkste Cipher Suite aus, die bei beiden Parteien verfügbar ist, und fügt sie dieser Nachricht hinzu.
3. Das *Certificate* wird vom Server gesendet. Damit wird dem Client ermöglicht, die Identität des Servers zu verifizieren.
4. *ServerHelloDone* signalisiert dem Client, dass der Server mit der Aushandlung der Verbindungsparameter abgeschlossen hat und jetzt auf eine Antwort vom Client wartet.

5. Mit dem public key aus dem Zertifikat prüft der Client die Authentizität des Servers. Er berechnet das premaster secret unter Zuhilfenahme von beiden Zufallszahlen. Es wird mit dem public Key des Servers verschlüsselt und bildet zusammen mit der Protokollversion die *ClientKeyExchange* Nachricht. Nur wenn der Server im Besitz des zum Zertifikat passenden private key ist, kann er die Nachricht entschlüsseln und den Verbindungsaufbau fortführen.
6. Durch *ChangeCipherSpec* wird der Server informiert, dass alle nachfolgenden Mitteilungen mit den ausgehandelten Parametern verschlüsselt sind.
7. *Finished* ist die erste verschlüsselte Nachricht des Verbindungsaufbaus. Sie enthält berechnete Hashwerte aus der gesamten bisherigen Kommunikation.
8. Der Server entschlüsselt mit seinem private key das premaster secret, und berechnet damit das master secret. Bei dieser Berechnung werden die zu Anfang generierten Zufallszahlen einbezogen. Weiterhin kann er anhand der empfangenen Hashwerte prüfen, ob es sich wirklich um den Client handelt, mit dem er den Aufbau der Verbindung gestartet hat. *ChangeCipherSpec* signalisiert wie beim Client die Verschlüsselung der folgenden Kommunikation.
9. Auch der Server bildet Hashwerte der vorangegangenen Nachrichten. Die *Finished* Nachricht wird wie beim Client mit dem master secret verschlüsselt.

Sollte während des Verbindungsaufbaus eine Prüfung seitens des Clients oder des Servers fehlschlagen, wird der Aufbau abgebrochen und die SSL Verbindung kommt nicht zustande.

2.2.2 Record Protokoll

Die vom Handshake Protokoll ausgehandelten Parameter werden vom Record Protokoll genutzt, um die Daten von der Applikationsschicht zu verschlüsseln. Dabei wird der Datenstrom auf der Senderseite zerlegt und auf der Empfängerseite wieder zusammengesetzt. Die Datenblöcke werden dabei mit Sequenznummern versehen und es erfolgt eine Kompression, wenn diese Option beim Verbindungsaufbau gewählt worden ist. Weiterhin wird bei der Fragmentierung ein Message Authentification Code (MAC) hinzugefügt.

2.3 DTLS

Das TLS Protokoll funktioniert nur auf einer Transportschicht, die Funktionen zur Sicherstellung der korrekten Datenübertragung bietet. Dazu gehören unter anderem die Erkennung von verloren gegangenen Paketen, die Sendungswiederholung und die Staukontrolle. Diese Mechanismen werden von TCP angeboten. Das UDP Protokoll hingegen kann diese Anforderungen nicht erfüllen.
Um die Kommunikation von Anwendungen abzusichern die UDP benutzen, wurde das Datagram Transport Layer Security Protokoll entwickelt Rescorla und Modadugu (2012). Es basiert weitgehend auf TLS, erweitert wurden lediglich die Funktionen, die für den Einsatz mit UDP nötig sind. Dabei handelt es sich im Prinzip um Mechanismen, die TCP anbietet, um Zuverlässigkeit herzustellen.

3 Analyse

In diesem Kapitel wird OpenSSL, dessen Kernkomponenten, sowie deren Funktionen vorgestellt. Es wird analysiert, mit welchen Ansätzen eine Portierung realisiert werden kann. Die Grundlage für diese Analyse bildet OpenSSL in der 1.0.1e vom 11. Februar 2013 OpenSSL.org (2013)

3.1 OpenSSL

Die in vielen Ländern geltenden Exportbeschränkungen erlauben den Export von Kryptographisoftware mit einer maximalen Schlüssellänge von lediglich 56Bit. Damit ist keine sichere Verschlüsselung möglich, heutige Hardware ermöglicht eine Dechiffrierung solch verschlüsselter Daten in kürzester Zeit. Dieser Umstand veranlasste den Australier Eric A. Young Mitte der Neunziger Jahre zusammen mit Tim Hudson SSLeay zu entwickeln. In Australien galten keine Exportbeschränkungen für Software die starke Verschlüsselung ermöglicht. Ende 1998 wechselten beide Entwickler zu RSA Security und ein neues Team übernahm die Weiterentwicklung von SSLeay unter dem Namen OpenSSL. Seitdem wird die Bibliothek von freiwilligen Entwicklern ständig weiterentwickelt.
Das OpenSSL Paket besteht im Grunde aus zwei Basisbibliotheken. Alle Kryptographie Funktionen wie symmetrische und asymmetrische Verschlüsselung, Hashfunktionen usw. befinden sich in der „crypto“-Bibliothek. Es sind viele Hilfsfunktionen enthalten, welche die Applikationsentwicklung mit OpenSSL erleichtern. Die „ssl“-Bibliothek bietet die nötigen Funktionen für die verschiedenen SSL Protokolle. Zusätzlich wird ein Programm mitgeliefert, welches auf der Kommandozeile die Funktionen der Bibliotheken nutzbar macht. Dazu gehören die Generierung von Schlüsselpaaren und Zertifikaten, Ver- und Entschlüsselung mit den angebotenen Verfahren, sowie ein SSL/TLS Testclient und Testserver. Im Folgenden werden die umfangreichen Hilfsfunktionen von OpenSSL aufgezeigt.

3.1.1 OpenSSL BIO-API

Ein wichtiger Bestandteil von OpenSSL ist die sogenannte BIO-API. Diese abstrahiert alle Eingabe- und Ausgabefunktionen und bündelt dabei sowohl verschlüsselte als auch unverschlüsselte Operationen. Vergleichbar ist diese Abstraktion mit der aus der Unix Welt bekannten Dateideskriptoren-Abstraktion. Dort werden Dateien und Sockets über dieselben „read()“- und „write()“-Funktionen benutzt. Dieses Modell der BIO-API ermöglicht damit dem Anwendungsentwickler die Arbeit mit verschlüsselten und unverschlüsselten Netzwerkverbindungen, ohne ständig zwischen der Unix-Socket-API und der OpenSSL API wechseln zu müssen. Objekte der BIO-API können von drei verschiedenen Typen sein:

- **Quellen-BIO**
  Mit Hilfe dieser Objekte kann aus Dateien oder Sockets gelesen werden.
- **Senken-BIO**
  Senken dienen als Ziel für Schreiboperationen.
- **Filter-BIO**
  Filter Objekte der BIO-API lassen Operationen zum Bearbeiten der Daten zu, das heißt, die Daten dieser Objekte können auf verschiedene Weisen manipuliert werden. Typische Filteroperationen sind die Chiffrierung und Dechiffrierung, aber auch die Umwandlung in andere Zeichenkodierungen ist zum Beispiel möglich.

Die drei Typen von BIO Objekten können miteinander verkettet werden. Einzige Bedingung bei der Bildung dieser Verkettungen ist, dass diese nur jeweils eine einzige Quelle und Senke aufweisen dürfen. Der Anzahl von Filter Objekten in einer Kette ist hingegen keine Grenze gesetzt. Eine typische Verkettung von BIO Objekten liest mithilfe eines Quellen-BIO Daten aus einer Datei oder einem Socket, verarbeitet diese Daten von einem oder mehreren Filter Objekten und schreibt diese dann in ein Senken Objekt.

3.1.2 OpenSSL ERR-API

Bei der OpenSSL-Bibliothek sind die Funktionen stark miteinander verschachtelt. Durch den hohen Abstraktionsgrad bleibt dieses in der Regel verborgen. Das erleichtert die Entwicklung von Applikationen mit der OpenSSL Bibliothek, wird bei der Fehlersuche allerdings zum Problem. Die Standard Library von C bietet über den <errno.h> Header Makros zur Fehlerbehandlung von aufgetretenen Fehlern an. Bei verschachtelten

Aufrufen kommt es hier allerdings zu einer Überlagerung, wenn Funktionen auf höheren Ebenen die Fehlercodes von unteren Ebenen überschreiben. Diesen Umstand umgeht OpenSSL, indem für die internen OpenSSL Funktionen Error-Queues erstellt werden. Für diese werden Hilfsfunktionen angeboten. Diese erlauben eine detaillierte Abarbeitung der Queues einschließlich der Ausgabe des genauen Ortes des Fehlers und der Daten, die zu diesem Zeitpunkt verarbeitet worden sind. Die Vielzahl an internen globalen Datenstrukturen erschwert einen Einsatz von OpenSSL in einer Multithread-Umgebung. Um diesen Umgang zu erleichtern werden Callback-Funktionen mitgeliefert. Die nötigen Werkzeuge zum Schutz kritischer Abschnitte und Ressourcen, wie Mutexe und Semaphoren, werden damit unabhängig von der Plattform einsetzbar.

3.1.3 Portierungsansatz

Eingebettete Systeme sind in ihren Ressourcen in der Regel sehr eingeschränkt. Das gilt für den Prozessor, den Arbeitsspeicher sowie den Flash Speicher. Der Flash Speicher muss das Betriebssystem und die Applikation aufnehmen. Das erzeugte Kompilat darf dessen Größe nicht überschreiten. Es muss also zuerst geprüft werden, ob die kompilierte Testanwendung, die Funktionen aus der OpenSSL Bibliothek benutzt, diese harte Restriktion erfüllt.
Die Testanwendung ist ein minimaler OpenSSL Server. Es handelt sich dabei um den modifizierten Quellcode aus (Viega u. a., 2002, S. 93ff.).

Der Quelltext des Servers:

```
1  #include <openssl/ssl.h>
2  #define PORT            "16000"
3  #define SERVER          "localhost"
4  #define CLIENT          "localhost"
5  #define int_error(msg)  handle_error(__FILE__, __LINE__, msg)
6
7  static unsigned char dh512_p[] = { 0xDA, 0x58, 0x3C, 0x16, 0xD9, 0x85, 0x22,
8  0x89, 0xD0, 0xE4, 0xAF, 0x75, 0x6F, 0x4C, 0xCA, 0x92, 0xDD, 0x4B, 0xE5, 0x33,
9  0xB8, 0x04, 0xFB, 0x0F, 0xED, 0x94, 0xEF, 0x9C, 0x8A, 0x44, 0x03, 0xED, 0x57,
10 0x46, 0x50, 0xD3, 0x69, 0x99, 0xDB, 0x29, 0xD7, 0x76, 0x27, 0x6B, 0xA2, 0xD3,
11 0xD4, 0x12, 0xE2, 0x18, 0xF4, 0xDD, 0x1E, 0x08, 0x4C, 0xF6, 0xD8, 0x00, 0x3E,
12 0x7C, 0x47, 0x74, 0xE8, 0x33, };
13 static unsigned char dh512_g[] = { 0x02, };
14
15 static DH *get_dh512(void) {
```

```
16          DH *dh = NULL;
17          if ((dh = DH_new()) == NULL )
18                  return (NULL );
19          dh->p = BN_bin2bn(dh512_p, sizeof(dh512_p), NULL );
20          dh->g = BN_bin2bn(dh512_g, sizeof(dh512_g), NULL );
21          if ((dh->p == NULL )|| (dh->g == NULL))return(NULL);
22          return (dh);
23  }
24
25  void handle_error(const char *file, int lineno, const char *msg)
26  {
27      fprintf(stderr, "** %s:%i %s\n", file, lineno, msg);
28      exit(-1);
29  }
30
31  void init_OpenSSL(void)
32  {
33      if (!SSL_library_init())
34      {
35          fprintf(stderr, "** OpenSSL initialization failed!\n");
36          exit(-1);
37      }
38       OpenSSL_add_all_algorithms();
39  }
40
41  SSL_CTX *setup_server_ctx(void) {
42          SSL_CTX *ctx;
43          DH *dh = NULL;
44          dh = get_dh512();
45          ctx = SSL_CTX_new( TLSv1_server_method() );
46          SSL_CTX_set_tmp_dh(ctx, dh);
47
48          if (SSL_CTX_set_cipher_list(ctx, "ADH-RC4-MD5") != 1) {
49                  int_error("Error setting cipher list (no valid ciphers)");
50          }
51          return ctx;
52  }
53
54  int do_server_loop(SSL *ssl) {
55          int err, nread;
56          char buf[80];
57
58          do {
59                  for (nread = 0; nread < sizeof(buf); nread += err) {
60                          err = SSL_read(ssl, buf + nread, sizeof(buf) - nread);
61                          if (err <= 0)
62                                  break;
63                  }
64                  printf("%s\n", buf);
65          } while (err > 0);
```

```
66          return (SSL_get_shutdown(ssl) & SSL_RECEIVED_SHUTDOWN) ? 1 : 0;
67  }
68
69  int main(int argc, char *argv[]) {
70          BIO *acc, *client;
71          SSL *ssl;
72          SSL_CTX *ctx;
73
74          init_OpenSSL();
75
76          ctx = setup_server_ctx();
77
78          acc = BIO_new_accept(PORT);
79          if (!acc) {
80                  int_error("Error creating server socket");
81          }
82
83          if (BIO_do_accept(acc) <= 0) {
84                  int_error("Error binding server socket");
85          }
86
87          for (;;) {
88                  if (BIO_do_accept(acc) <= 0) {
89                          int_error("Error accepting connection");
90                  }
91
92                  client = BIO_pop(acc);
93                  if (!(ssl = SSL_new(ctx))) {
94                          int_error("Error creating SSL context");
95                  }
96                  SSL_set_accept_state(ssl);
97                  SSL_set_bio(ssl, client, client);
98
99                  if (SSL_accept(ssl) <= 0) {
100                         int_error("Error accepting SSL connection");
101                 }
102
103                 fprintf(stderr, "SSL Connection opened\n");
104                 if (do_server_loop(ssl)) {
105                         SSL_shutdown(ssl);
106                 }
107                 else {
108                         SSL_clear(ssl);
109                 }
110                 fprintf(stderr, "SSL Connection closed\n");
111                 SSL_free(ssl);
112         }
113         SSL_CTX_free(ctx);
114         BIO_free(acc);
115         return 0;
```

```
116 }
```

Der Funktionsumfang des Servers ist bewusst minimal gehalten. Er initialisiert die OpenSSL Bibliothek und erstellt einen SSL Kontext. In diesem Kontext bietet er lediglich das TLSv1.1 Protokoll an. Ebenso stellt er dem Client nur eine Cipher Suite zur Auswahl: Anonymes Diffie-Hellman-Protokoll für den Schlüsselaustausch, RC4 für die Verschlüsselung und MD5 als Hashfunktion. Der nötige Socket wird erstellt und bei der Verbindungsanfrage eines Clients wird der SSL Handshake mit den Eigenschaften des Kontext initiiert. Wenn der Handshake erfolgreich durchgeführt werden konnte beginnt der verschlüsselte Datenaustausch.

Die Kompilierung der Testanwendung resultiert in einem Abbild für den Flash Speicher von 1,7 MiB. Das MSB-A2 hat einen 512 KiB großen Flashbaustein. Der Einsatz dieser Serveranwendung auf dem Board ist damit nicht möglich, weil die Codegröße zu umfangreich ist. Die Portierung kann nur durchgeführt werden, wenn eine Möglichkeit der Reduzierung gefunden wird. Ein Ansatz ist die bedingte Kompilierung.

3.1.3.1 Bedingte Kompilierung

Um die Größe des Kompilats zu reduzieren wird bedingte Kompilierung eingesetzt. Durch bedingte Kompilierung werden Bereiche des Quellcodes vom Compiler von der Übersetzung ausgenommen. Daraus resultiert, dass der Applikation diese Funktionen nicht mehr zur Verfügung stehen. Die Möglichkeiten bei der OpenSSL Bibliothek reichen von der Entfernung einzelner SSL Protokolle und Cipher Suites bis hin zur Abschaltung der Unterstützung für spezielle Kryptographie Hardware.

Der Ansatz ist jetzt, alle Funktionen, die nicht von der Testanwendung benutzt werden, von der Kompilierung auszuschließen. Die OpenSSL Bibliothek wird dafür mit den folgenden Compiler Optionen neu übersetzt:

no-idea no-camellia no-seed no-bf no-cast no-des no-rc2 no-rc5 no-md2 no-md4 no-ripemd no-mdc2 no-rsa no-dsa no-ecdsa no-ecdh no-ssl2 no-ssl3 no-krb5 no-engine no-hw no-tlsext no-cms no-jpake no-capieng no-srtp

Das Ergebnis des Ansatzes reduziert die Größe der statischen Bibliotheken. Die crypto Bibliothek wird von 3,1 auf 1,7 MiB verkleinert, die Größe der ssl Bibliothek verringert sich von 532 auf 250 KiB. Die Testanwendung wird nun mit den reduzierten Bibliotheken

neu kompiliert. Es stellt sich heraus, dass diese Reduzierungen keine Auswirkungen auf die Größe der Testanwendung hat. Dieses Ergebnis wird im folgenden Abschnitt analysiert.

3.1.3.2 Analyse des Ansatzes

Eine statische Bibliothek ist ein Archiv, das aus den vom Compiler generierten Objektdateien besteht. Beim Kompilieren einer Anwendung bezieht der Linker die von der Anwendung benötigten Objektdateien und erstellt damit das ausführbare Programm. Dabei wird bereits bei dem Aufruf einer einzelnen Funktion aus einem Objekt die gesamte Datei inkludiert.

Nach einer Analyse der Struktur und des Quelltextes der OpenSSL Bibliothek stellt sich heraus, dass deren Komponenten stark miteinander verschachtelt sind. Der Grund für diese Verschachtelungen ist nicht immer offensichtlich. Diese Eigenschaft von OpenSSL wird hier an einem Beispiel verdeutlicht.
Die Testanwendung benötigt aufgrund des anonymen Diffie-Hellmann-Protokolls keinerlei Funktionen für den Umgang mit Zertifikaten. Trotzdem wird bei ihrer Kompilierung die Objektdatei für die X.509 Infrastruktur vom Linker inkludiert. Der Grund ist der in Zeile 50 des Quelltextes erstellte SSL Kontext. OpenSSL benutzt intern die ASN.1 Beschreibungssprache für alle benötigten Datenstrukturen. Bei der Erstellung des Kontextes werden daher die Funktionen für ASN.1 aufgerufen und müssen vom Linker mit einbezogen werden. Die ASN.1 Implementierung wiederum referenziert die X.509 Infrastruktur, was auch deren Inkludierung erklärt. Diese Verschachtelungen finden sich häufig in der Bibliothek, was sich direkt auf die Größe der Applikation auswirkt.

Die Analyse macht deutlich, dass die OpenSSL Bibliothek für eine Portierung auf eingebettete System ungeeignet ist. Schon die minimale Testanwendung konnte die harte Restriktion an die Codegröße nicht einhalten. Der Ansatz der bedingten Kompilierung konnte dieses Problem nicht lösen.

Um der Zielsetzung dieser Arbeit noch gerecht zu werden, werden im folgenden Abschnitt alternative SSL Implementierungen betrachtet.

3.2 Evaluierung von OpenSSL Alternativen

Nachdem die Portierung von OpenSSL nicht erfolgreich durchgeführt werden konnte, werden jetzt mögliche Alternativen evaluiert. Dabei werden nur Open Source SSL Implementierungen betrachtet. Das Hauptkriterium bei der Betrachtung ist der Codeumfang der Lösung. Auf diesen muss besonderen Wert gelegt werden, wie im letzten Abschnitt deutlich gemacht worden ist. Weitere Kriterien sind die unterstützten Protokolle, eventuelle Abhängigkeiten der Bibliothek und die Vorbereitungen, die getroffen worden sind, um eine Portierung zu ermöglichen. Lösungen, die abhängig von Funktionen sind, die RIOT nicht anbieten kann, werden von der Evaluierung ausgeschlossen. Beispiele für solche sind die Abhängigkeit von Java oder C++ 11.

	LoC	Abhängigk.	Protokolle	Portierungsvorb.	Besonderh.
GnuTLS	180k	libnettle & gmplib	komplett	vorhanden	-
CyaSSL	64k	send()/recv() API	komplett	vorhanden	-
MatrixSSL	21k	-	komplett	sehr gut	FIPS 140-2
PolarSSL	46k	send()/recv() API	kein DTLS	vorhanden	-

Die Anzahl der Lines of Code wurde mit CLOC CLOC (2013) Version 1.6 ermittelt.

3.3 Ergebnis der Evaluierung

PolarSSL bietet keine Unterstützung für das in im Umfeld von WSNs wichtige DTLS Protokoll, damit kommt es für die Portierung nicht in Frage.
GnuTLS ist abhängig von den Bibliotheken libnettle und gmplib, außerdem hat es den mit Abstand größten Codeumfang.
CyaSSL und MatrixSSL sind beide gut für eine Portierung geeignet. Da MatrixSSL den geringeren Codeumfang und keinerlei Abhängigkeiten besitzt, wird diese Implementierung für die Portierung verwendet.

4 Portierung

Dieses Kapitel beschreibt die nötigen Schritte, um die im letzten Kapitel ausgewählte MatrixSSL Bibliothek zu portieren. Dafür wird ein Einblick in die Anwendungsentwicklung mit dem RIOT Betriebssystem gegeben und die Zielplattform vorgestellt. Es wird ein kurzer Einblick in die Konfiguration von MatrixSSL gegeben, gefolgt von den Schritten, welche zur Portierung auf RIOT nötig sind.

4.1 MSB-A2

In dieser Arbeit wird das Scatterweb MSB-A2 Baar u. a. (2008) Entwicklungsboard als Zielplattform für die Portierung der SSL Bibliothek verwendet. Scatterweb ist eine Plattform für selbst-konfigurierende Wireless Sensor Networks, das als Forschungsprojekt an der FU Berlin gestartet worden ist. Das MSB-A2 ist die Weiterentwicklung des MSB-430, das noch eine 16 Bit MCU enthielt.

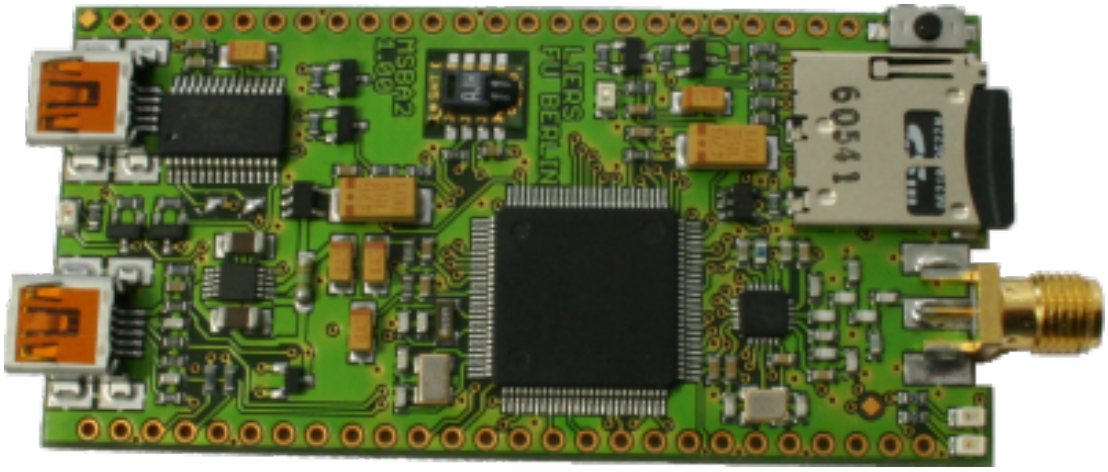

Abb. 4.1: MSB-A2 Entwicklungsboard MSB-A2 (2013)

Wesentliche Daten des MSB-A2:

- NXP LPC2387 Mikrocontroller mit ARM7TDMI-S Core, bis zu 72MHz
- Funkmodul CC1100 von Texas Instruments
- 512 KiB ROM
- 98 KiB RAM
- Real time clock
- Vectored Interrupt Controller
- Keine MMU

Das Board wurde mit Sicht auf minimale Kosten entwickelt, um es bezahlbar zu halten. Es hat deswegen nur einen Mikrocontroller, ein Funkmodul und die nötigen Interfaces zum Programmieren und Debuggen. Durch den modularen Aufbau kann es um weitere Sensoren und Aktoren erweitert werden. Alle nicht verbundenen Interfaces und I/O Pins sind über eine Sockelleiste herausgeführt worden. Durch dieses Designkonzept ist es möglich, die Plattform mit wenig Aufwand um weitere Erweiterungen auszustatten.

4.2 RIOT-OS

RIOT-OS riot os.org (2013) ist ein Betriebssystem für Wireless Sensor Networks. Die Entwicklung von RIOT wurde 2008 gestartet, damals noch unter dem Namen Feuerware. Seitdem wird es kontinuierlich um neue Funktionen und Protokolle erweitert, 2010 wurde es zu ukleos geforked. Seit 2013 nennt sich das Betriebssystem RIOT. Es ist mittlerweile teilweise POSIX kompatibel, mit dem Ziel, eine vollständige Kompatibilität zu erreichen. Der große Vorteil zu verbreiteten Betriebssystemen für Knoten eines WSNs ist, dass zur Applikationsentwicklung Standard-Werkzeuge wie gcc und gdb verwendet werden können. Dadurch wird eine große Entwicklergemeinde angesprochen, da diese sich nicht bei Entwicklung für RIOT von ihrer vertrauten Umgebung umgewöhnen müssen. RIOT bietet sowohl Unterstützung für Multi-Threading als auch Real-Time Anwendungen und ist dabei nicht auf eine Memory Control Unit angewiesen. Die Anzahl der unterstützten Hardware-Plattformen wächst dabei stetig. Zu ihnen gehören:

- MSP430

- MSB-A2
- Texas Instruments EZ430-Chronos
- STM32F4DISCOVERY

Bei der Entwicklung des Betriebssystems wurde streng auf die Restriktionen von Knoten eines WSNs geachtet. Es basiert auf einer Mikrokernel Architektur, die zusammen mit einem tickless scheduler einen sehr sparsamen Energiehaushalt ermöglicht.

4.2.1 RIOT-Entwicklung

Bei der Entwicklung von Applikationen für eingebettete Systeme wird in der Regel ein Crosscompiler benutzt. Crosscompiler erzeugen Binärcode für eine bestimmte Zielarchitektur, unabhängig auf welchem Hostsystem sie eingesetzt werden. Dadurch werden die Entwicklungszyklen deutlich verkürzt, ein Kompilierungsdurchlauf benötigt nur einen Bruchteil der Zeit, die direkt auf dem Zielsystem nötig gewesen wäre. Von den RIOT Entwicklern wird die CodeBench von Mentor Graphics MentorGraphics (2013) empfohlen, die auch den nötigen ARM Crosscompiler enthält. Die Codebench ist für Linux, Windows und Mac OS X verfügbar.
Nach der Kompilierung wird der Binärcode des RIOT Systems und der Applikation in einer Datei zusammengefasst. Die Datei ist das Abbild des Flashspeichers der Zielplattform und darf dessen Größe nicht überschreiten. Das Abbild wird auf den Flashspeicher geschrieben und RIOT startet zusammen mit der entwickelten Applikation auf dem Entwicklungsboard.

4.2.1.1 Native Plattform

Eine Sonderstellung nimmt die native Plattform in der RIOT Distribution ein. Mit ihrer Hilfe wird RIOT als Prozess auf einem Linux Hostsystem gestartet. Diese Plattform erleichtert die Applikationsentwicklung in vielerlei Hinsicht. Die Netzwerkgeräte und Protokolle des Hosts stehen damit zur Verfügung. Sie ermöglichen den Test von Funktionen, die mit dem aktuellen Entwicklungsstand von RIOT im Netzwerkbereich noch nicht möglich wären. Weiterhin entfällt das Beschreiben des Flashspeichers für den Test der Anwendung nach der Kompilierung. Der größte Vorteil der native Plattform bietet sich beim Debugging. Die Entwicklungsplattformen können nur über USB oder serielle Verbindungen Debug-Ausgaben liefern. Einen Prozess auf einem Hostsystem zu untersuchen eröffnet deutlich mehr Möglichkeiten.

4.3 Matrix SSL

4.3.1 Konfiguration

Die gesamte Bibliothek wird über #defines in den Header-Dateien konfiguriert. Diese erlauben das Aktivieren von Funktionen und die Steuerung des Laufzeitverhaltens. Durch diesen Lösungsansatz werden nur Teile in die statische Bibliothek kompiliert, die auch wirklich genutzt werden. Diese bedingte Kompilierung ermöglicht es, eine minimale Codegrösse zu erreichen. Im Folgenden werden die wichtigsten #defines aus den Konfigurationsdateien aufgezeigt.

coreConfig.h
Hier werden globale Einstellungen für das Laufzeitverhalten der Bibliothek vorgenommen. So wird das Verhalten im Fehlerfall gesteuert und der Umfang der Debug-Ausgaben angepasst.

matrixsslConfig.h
In dieser Header-Datei werden die Cipher Suites gewählt, die zum Aufbau einer SSL Verbindung zur Verfügung stehen sollen. Auch die möglichen Protokollversionen von SSL können hier konfiguriert werden.

cryptoConfig.h
Die angebotenen Kryptographiefunktionen werden hier aktiviert. Dazu zählen die symmetrischen Verfahren sowie die Hashfunktionen. Weiterhin werden die Standards für die Speicherung der öffentlichen Schlüssel und Zertifikate gewählt.

4.3.2 Portierung

Die eigentliche Portierung besteht in der Implementierung der plattformabhängigen Funktionen. Zu diesen zählen die Zeitfunktionen, die Funktion zum Generieren von Zufallszahlen sowie, falls benötigt, Programmcode, der einen Zugriff auf das Dateisystem der Plattform ermöglicht. Letzteres ist bei der Portierung auf RIOT nicht möglich, weil der aktuelle Entwicklungsstand des Betriebssystems noch keine derartigen Funktionalitäten zur Verfügung stellt. Damit ist der SSL Implementierung der Zugriff auf in Dateien gespeicherten Schlüsseln und Zertifikaten verwehrt. Diese können alternativ auch in Header-Dateien abgelegt werden. Das verringert die Flexibilität, weil bei einem

Austausch die Bibliothek neu übersetzt werden muss, ermöglicht dadurch allerdings die Portierung auf RIOT.
Alle Funktionen werden zentral in der Datei osdep.c implementiert, und im nachfolgenden Teil beschrieben.

4.3.2.1 Zeitfunktionen

Es existieren mehrere Gründe, warum Zeitfunktionen essenziell notwendig sind. Beim SSL Handshake werden die ClientHello und ServerHello Nachrichten mit Zeitstempeln versehen. Wenn die Verbindung aufgebaut ist, muss der Server die Session verwalten. In einem Multithreaded Umfeld kann die Anzahl der Sessions umfangreich sein. Um erkennen zu können, welche Session abgelaufen bzw. blockiert ist, ist der Server auf eine Zeitbasis angewiesen. Der Zugriff auf die Echtzeituhr ist bei jedem System unterschiedlich und muss implementiert werden. Das RIOT Modul rtc bietet die nötigen Funktionen an und muss bei der Kompilierung im Makefile aktiviert werden.

```
int osdepTimeOpen(void);
```

Initialisiert und startet die Echtzeituhr.

```
PSPUBLIC int osdepTimeOpen(void) {
        printf("\n\nInitializing␣RTC...\n");
        rtc_init();
        rtc_reset();
        rtc_enable();
        printf("RTC␣init␣done.\n\n");
        return PS_SUCCESS;
}
```

```
int32 psGetTime(psTime_t *currentTime);
```

Schreibt die aktuelle Zeit in die übergebene Struktur currentTime.

```
PSPUBLIC int32 psGetTime(psTime_t *t) {
        rtc_time(t);
        return t->tv_sec;
}
```

```
int32 psDiffMsecs(psTime_t then, psTime_t now);
```

Liefert die Differenz zwischen zwei übergebenen Zeitstrukturen zurück. An dieser Stelle wird die teilweise vorhandene POSIX Kompatibilität von RIOT ausgenutzt. Die Funktionen können von der POSIX Implementierung von MatrixSSL übernommen werden.

```
int32 psCompare(psTime_t a, psTime_t b);
```

Diese Funktion ermittelt, welche von beiden Zeitstrukturen ein älteres Datum enthält. Auch hier kann der POSIX Programmcode verwendet werden.

4.3.2.2 Zufallsgenerator

Sicher generierte Zufallszahlen sind sowohl ein elementarer Bestandteil für das SSL Protokoll, als auch für die dabei verwendeten kryptographischen Algorithmen. Zur Verwendung der Funktionen zum Generieren von Zufallszahlen muss das Modul random inkludiert werden.

```
int osdepEntropyOpen(void);
```

Zum Initialisieren des PRNG wird diese Funktion benutzt. Als minimaler Test der erfolgreichen Initialisierung werden 64 Zufallszahlen generiert. Keine der Zahlen darf kleiner oder gleich null sein, ansonsten gilt die Initialisierung als fehlgeschlagen.

```
PSPUBLIC int osdepEntropyOpen(void) {
        printf("Initializing Random Generator... \n");
        uint32_t init[4] = { 0x714, 0x244, 0x963, 0x788 }, length = 4;
        genrand_init_by_array(init, length);
        int i;
        while(i++ < 64) {
                if ( genrand_uint32() <= 0 ) {
                        printf("PRNG init failed!\n");
                        return PS_FAILURE;
                }
        }
        printf("Random Generator init successful.\n\n");
        return PS_SUCCESS;
}
```

```
int32 psGetEntropy(unsigned char *bytes, uint32 size);
```

Die Funktion generiert eine Anzahl von Zufallszahlen, und schreibt sie in den übergebenen Puffer.

```
PSPUBLIC int32 psGetEntropy(unsigned char *bytes, uint32 size) {
        while(size--- > 0) {
                *(bytes++) = genrand_uint32();
}
        return PS_SUCCESS;
}
```

Hier wird deutlich, dass die von MatrixSSL getroffene Vorbereitung auf eine Portierung sehr gut gelungen ist. Die plattformabhängigen Funktionen sind auf ein Minimum reduziert und streng gekapselt worden.

5 Test

Sobald alle in 4.3.2 beschriebenen plattformabhängigen Funktionen implementiert worden sind, kann mit der Validierung begonnen werden, um die korrekte Funktionsweise zu verifizieren. Dazu werden die von MatrixSSL mitgelieferten Testprogramme genutzt.

5.1 SSLTest

Das Programm SSLTest prüft in einem ersten Schritt, ob ein SSL Handshake erfolgreich durchgeführt werden kann. Es verzichtet dabei auf die Nutzung von Sockets. Zusätzlich wird eine Zeitmessung durchgeführt. Variabel einzustellen ist die Anzahl der Testdurchläufe sowie die Menge der auszutauschenden Nutzdaten. Dabei geht der Austausch dieser nicht in die Zeitmessung ein.

Der Test wird mit zehn Iterationen und einem KiB Daten durchgeführt:

```
#define CONN_ITER               10      /* number of connections per type of hs */
#define APP_DATA_LEN            1024
```

Die Ausgabe von SSLTest:

```
Reset CPU (into user code)
Board initialized.
kernel_init(): This is RIOT!
Scheduler...[OK]
kernel_init(): jumping into first task...


Initializing RTC...
RTC init done.

Initializing Pseudo Random Generator...
Pesudo Random Generator init done.

Testing TLS_RSA_WITH_AES_256_CBC_SHA suite
        Standard handshake test
                10 connections
                CLIENT: 189 msecs/connection
```

```
                SERVER: 1136 msecs/connection

Testing TLS_RSA_WITH_AES_128_CBC_SHA suite
        Standard handshake test
                10 connections
                CLIENT: 189 msecs/connection
                SERVER: 1139 msecs/connection

Testing SSL_RSA_WITH_3DES_EDE_CBC_SHA suite
        Standard handshake test
                10 connections
                CLIENT: 188 msecs/connection
                SERVER: 1138 msecs/connection

Testing SSL_RSA_WITH_RC4_128_SHA suite
        Standard handshake test
                10 connections
                CLIENT: 189 msecs/connection
                SERVER: 1138 msecs/connection

Testing SSL_RSA_WITH_RC4_128_MD5 suite
        Standard handshake test
                10 connections
                CLIENT: 189 msecs/connection
                SERVER: 1139 msecs/connection
```

Dieser erfolgreiche Durchlauf von SSLTest zeigt, dass die Portierung korrekt durchgeführt worden ist. Um die Zeitmessung zu plausibilisieren wurde MatrixSSL zusätzlich auf eine weitere ARM Plattform portiert, der Marvell Kirkwood 88F6281 Marvell (2013). Das Board besitzt einen 1200 MHz Prozessor und wird mit dem Linux Betriebssystem betrieben. Die Ergebnisse sind durch die verschiedene Umgebung und Hardware nicht direkt vergleichbar, geben aber einen Anhaltspunkt, ob die gemessenen Zeiten im erwarteten Bereich sind. Im folgenden die Ergebnisse der Kirkwood Plattform:

```
Testing TLS_RSA_WITH_AES_256_CBC_SHA suite
        Standard handshake test
                10 connections
                CLIENT: 8 msecs/connection
                SERVER: 49 msecs/connection

Testing TLS_RSA_WITH_AES_128_CBC_SHA suite
        Standard handshake test
                10 connections
                CLIENT: 8 msecs/connection
                SERVER: 49 msecs/connection

Testing SSL_RSA_WITH_3DES_EDE_CBC_SHA suite
        Standard handshake test
```

```
                10 connections
                CLIENT: 8 msecs/connection
                SERVER: 49 msecs/connection

Testing SSL_RSA_WITH_RC4_128_SHA suite
        Standard handshake test
                10 connections
                CLIENT: 8 msecs/connection
                SERVER: 49 msecs/connection

Testing SSL_RSA_WITH_RC4_128_MD5 suite
        Standard handshake test
                10 connections
                CLIENT: 8 msecs/connection
                SERVER: 49 msecs/connection
```

Der Leistungsfaktor der Prozessoren unterscheidet sich zwischen beiden Plattformen etwa um den Faktor 20. Hier wird deutlich, dass die ermittelten Werte des MSB-A2 plausibel sind.

5.2 Client/Server Test

Der erfolgreiche Testdurchlauf von SSLTest hat gezeigt, dass ein Handshake lokal auf dem MSB-A2 durchgeführt werden kann. In einem zweiten Schritt wird jetzt getestet, ob dieser auch zwischen zwei Instanzen über eine Netzwerkschnittstelle möglich ist. MatrixSSL liefert dafür eine Client/Server Applikation mit, die diesen Test ermöglicht. Zum Zeitpunkt dieser Arbeit sind die Implementierungen für die nötigen TCP/UDP Protokolle noch nicht vollständig abgeschlossen. Der Client/Server Test wird aus diesem Grund mit der in 4.2.1.1 beschriebenen native Plattform durchgeführt.

5.2.1 Serverausgabe

```
RIOT native cpu initialized.
RIOT native interrupts/signals initialized.
RIOT native tap initialized.
RIOT native uart0 initialized.
LED_GREEN_OFF
LED_RED_ON
RIOT native board initialized.
RIOT native hardware initialization complete.
kernel_init(): This is RIOT!
Scheduler...[OK]
kernel_init(): jumping into first task...
```

```
UART0 thread started.
uart0_init() [OK]
native rtc initialized
Native LTC4150 initialized.
Main started!
Listening on port 16000
RECV PARSED: [GET / HTTP/1.0]
RECV PARSED: [User-Agent: MatrixSSL/3.4.2-OPEN]
RECV PARSED: [Accept: */*]
RECV PARSED: [Content-Length: 0]
RECV COMPLETE HTTP MESSAGE
RECV PARSED: [GET / HTTP/1.0]
RECV PARSED: [User-Agent: MatrixSSL/3.4.2-OPEN]
RECV PARSED: [Accept: */*]
RECV PARSED: [Content-Length: 0]
RECV COMPLETE HTTP MESSAGE
```

5.2.2 Clientausgabe

```
RIOT native cpu initialized.
RIOT native interrupts/signals initialized.
RIOT native tap initialized.
RIOT native uart0 initialized.
LED_GREEN_OFF
LED_RED_ON
RIOT native board initialized.
RIOT native hardware initialization complete.

kernel_init(): This is RIOT!
Scheduler...[OK]
kernel_init(): jumping into first task...
UART0 thread started.
uart0_init() [OK]
native rtc initialized
Native LTC4150 initialized.
=== INITIAL CLIENT SESSION ===
Validated cert for: Sample Matrix RSA-1024 Certificate.
SEND: [GET / HTTP/1.0
User-Agent: MatrixSSL/3.4.2-OPEN
Accept: */*
Content-Length: 0

]
RECV PARSED: [HTTP/1.0 200 OK]
RECV PARSED: [Server: MatrixSSL/3.4.2-OPEN]
RECV PARSED: [Pragma: no-cache]
RECV PARSED: [Cache-Control: no-cache]
RECV PARSED: [Content-type: text/plain]
RECV PARSED: [Content-length: 9]
HTTP data parsing not supported, ignoring.
```

```
32 RECV COMPLETE HTTP MESSAGE
33 HTTP data parsing not supported, ignoring.
34 SUCCESS: Received HTTP Response
35
36 === CLIENT SESSION WITH CACHED SESSION ID ===
37 SEND: [GET / HTTP/1.0
38 User-Agent: MatrixSSL/3.4.2-OPEN
39 Accept: */*
40 Content-Length: 0
41
42 ]
43 RECV PARSED: [HTTP/1.0 200 OK]
44 RECV PARSED: [Server: MatrixSSL/3.4.2-OPEN]
45 RECV PARSED: [Pragma: no-cache]
46 RECV PARSED: [Cache-Control: no-cache]
47 RECV PARSED: [Content-type: text/plain]
48 RECV PARSED: [Content-length: 9]
49 HTTP data parsing not supported, ignoring.
50 RECV COMPLETE HTTP MESSAGE
51 HTTP data parsing not supported, ignoring.
52 SUCCESS: Received HTTP Response
```

Der Test beweist, dass das Zertifikat des Servers validiert werden konnte. Ebenfalls konnte die Wiederaufnahme einer Session nachgewiesen werden.

Mit dem erfolgreichen Abschließen beider Testszenarien kann die Portierung als fehlerfrei betrachtet werden.

6 Zusammenfassung & Ausblick

In diesem Kapitel werden die Schritte der Arbeit zusammengefasst und es wird ein Ausblick auf die weitere Entwicklung im Bereich von WSNs gegeben.

6.1 Zusammenfassung

Die Zielsetzung dieser Arbeit war es, den Einsatz von Verschlüsselung auf dem RIOT Betriebssystems zu ermöglichen. Um diese Zielsetzung zu erfüllen, sollte die OpenSSL Implementierung portiert werden. In der Analyse hat sich herausgestellt, dass diese für eine Portierung ungeeignet ist. Aus diesem Grund wurden alternative Implementierungen evaluiert und die MatrixSSL Bibliothek wurde für die Portierung ausgewählt. Die nötigen Schritte für eine Portierung wurden beschrieben und mit Tests die korrekte Funktion nachgewiesen.
Diese Zielsetzung konnte erreicht werden, allerdings nicht mit der gewünschten OpenSSL Implementierung, sondern mit der MatrixSSL. Für Geräte mit wenig Speicher, wie sie in WSNs verwendet werden, ist eine für wenig Speicher optimierte SSL Lösung erforderlich. In dieser Arbeit wurde gezeigt, dass Lösungen, wie OpenSSL keine Alternative sind, da ihr Speicherbedarf für Knoten eines WSNs zu hoch ist.

6.2 Ausblick

Wireless Sensor Networks werden in Zukunft eine immer größere Rolle spielen. Aufgrund der hohen Stückzahl an Geräten in diesen Netzen, wird sich an der Ausstattung der Geräte nicht viel verändern. Ebenfalls werden sie zunehmend in sicherheitskritischen Bereichen Verwendung finden. Somit ist zu erwarten, dass die Anforderung an sichere Verbindungen der Geräte untereinander, stetig steigen wird. Da solche Verbindungen nicht mit Standardbibliotheken wie OpenSSL zu erreichen sind, werden sich in diesem Bereich embedded Bibliotheken, wie MatrixSSL durchsetzen.

Literaturverzeichnis

[NIST197 2001] *FIPS PUB 197, Advanced Encryption Standard (AES).* 2001. – U.S.Department of Commerce/National Institute of Standards and Technology

[Baar u. a. 2008] Baar, Michael ; Will, Heiko ; Blywis, Bastian ; Hillebrandt, Thomas ; Liers, Achim ; Wittenburg, Georg ; Schiller, Jochen: The ScatterWeb MSB-A2 Platform for Wireless Sensor Networks / Freie Universität Berlin, Department of Mathematics and Computer Science, Institute for Computer Science, Telematics and Computer Systems group. Takustraße 9, 14195 Berlin, Germany, 09 2008 (TR-B-08-15). – Forschungsbericht. – URL ftp://ftp.inf.fu-berlin.de/pub/reports/tr-b-08-15.pdf

[CLOC 2013] CLOC: *CLOC.* http://cloc.sourceforge.net. 2013. – [Online; Stand 29. November 2013]

[codeproject 2013] codeproject: *SSL Handshake.* 2013. – URL http://www.codeproject.com/KB/IP/326574/1WaySSL.png. – [Online; Stand 22. Oktober 2013]

[Des 1977] Des: Data Encryption Standard. In: *In FIPS PUB 46, Federal Information Processing Standards Publication*, 1977, S. 46–2

[Dierks und Allen 1999] Dierks, Tim ; Allen, Christopher: The TLS Protocol Version 1.0 / IETF. January 1999 (2246). – RFC

[Eastlake und Jones 2001] Eastlake, D. ; Jones, P.: US Secure Hash Algorithm 1 (SHA1) / IETF. September 2001 (3174). – RFC

[Marvell 2013] Marvell: *Marvell 88F6281.* 2013. – URL http://www.marvell.com/embedded-processors/kirkwood/assets/HW_88F6281_OpenSource.pdf. – [Online; Stand 22. Oktober 2013]

[MentorGraphics 2013] MentorGraphics: *CodeBench 2008q3.* http://www.codesourcery.com/sgpp/lite/arm/portal/release642. 2013. – [Online; Stand 28. November 2013]

[Montenegro u. a. 2007] Montenegro, G. ; Kushalnagar, N. ; Hui, J. ; Culler, D.: Transmission of IPv6 Packets over IEEE 802.15.4 Networks / IETF. September 2007 (4944). – RFC

[MSB-A2 2013] MSB-A2: *MSB-A2 Entwicklungsboard*. 2013. – URL http://www.des-testbed.net/sites/default/files/MSB-A2.png. – [Online; Stand 22. Oktober 2013]

[OpenSSL.org 2013] OpenSSL.org: *OpenSSL 1.0.1e*. http://www.openssl.org/source/openssl-1.0.1e.tar.gz. 2013

[riot os.org 2013] os.org riot: *RIOT-OS*. http://riot-os.org. 2013

[Postel 1981] Postel, Jon: Transmission Control Protocol / IETF. September 1981 (793). – RFC

[Rescorla und Modadugu 2012] Rescorla, E. ; Modadugu, N.: Datagram Transport Layer Security Version 1.2 / IETF. January 2012 (6347). – RFC

[Rivest u. a. 1978] Rivest, R. L. ; Shamir, A. ; Adleman, L.: A Method for Obtaining Digital Signatures and Public-key Cryptosystems. In: *Commun. ACM* 21 (1978), Feb., Nr. 2, S. 120–126

[Rivest 1992] Rivest, Ronald L.: The MD5 Message-Digest Algorithm / IETF. April 1992 (1321). – RFC

[Viega u. a. 2002] Viega, Jon ; Chandra, Pravir ; Messier, Matt: *Network Security with Openssl*. 1st. Sebastopol, CA, USA : O'Reilly & Associates, Inc., 2002. – ISBN 059600270X

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 02. Dezember 2013 Sascha Klaholz