Universität Bremen

Fachbereich 3: Mathematik und Informatik

Bachelorarbeit

Anpassungen von DTLS zur sicheren Kommunikation in eingeschränkten Umgebungen

Lars Schmertmann

Matrikel-Nr. 2469187

30. September 2013

1. Gutachter: Prof. Dr.-Ing. Carsten Bormann
2. Gutachter: Dr.-Ing. Olaf Bergmann
Betreuer: Dr.-Ing. Olaf Bergmann

Lars Schmertmann

Anpassungen von DTLS zur sicheren Kommunikation in eingeschränkten Umgebungen

Bachelorarbeit, Fachbereich 3: Mathematik und Informatik

Universität Bremen, September 2013

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Bremen, den 30. September 2013

Lars Schmertmann

Danksagung

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln. Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

Olaf Bergmann - Unterstützung 1

Jens Trillmann - Unterstützung 2

Dominik Menke - Vorlage 3

Zusammenfassung

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

Inhaltsverzeichnis

List of TODOs

remove \listoftodos again

Kapitel 1

Einleitung

1.1 Motivation

Während es im Internet schon seit Mitte der 90er Jahre das „Transport Layer Security (TLS) Protocol“ [DR08] (früher SSL) gibt, um den Datenverkehr über das „Transmission Control Protocol (TCP)“ abzusichern fehlte lange Zeit ein Standard um den Datenverkehr über das „User Datagram Protocol (UDP)“ zu sichern, während die Beliebtheit des Protokolls, unter anderem im Bereich der Onlinespiele, zunahm [RM12, Kapitel 1]. Um diese Lücke zu schließen begann die Internet Engineering Task Force (IETF) 2004 damit ein Protokoll nach dem Vorbild von TLS zu entwickeln, was 2006 schließlich zu der Standardisierung des „Datagram Transport Layer Security (DTLS) Protocol“ [RM12] führte. Dieses ist fast identisch mit TLS, wurde jedoch um Mechanismen ergänzt, die in UDP im Gegensatz zu TCP fehlen. Dazu gehört insbesondere die Zuverlässigkeit der Datenübertragung die beim Verbindungsaufbau und somit der Aushandlung der Sicherheitsmechanismen notwendig ist. TLS und DTLS haben sich im Internet bewährt, sind jedoch zu einer Zeit entstanden, als das „Web of Things (WoT)“ noch nicht vertreten war. Die dort verwendeten Geräte haben nur sehr wenig Ressourcen zur Verfügung. Dies betrifft neben wenig Rechenleistung und Speicher auch den Energievorrat. Etabliert hat sich dort das „Constrained Application Protocol (CoAP)“ [She+12] über UDP aufgrund seines schlanken Designs. Passend zu UDP ist das umfangreiche DTLS-Protokoll, das sich jedoch nicht so einfach auf „kleinen“ Endgeräten realisieren lässt. Genau hier soll diese Arbeit ansetzen und DTLS entsprechend Anpassen damit es auch auf Geräten mit wenig Ressourcen funktionieren kann.

1.2 Verwandte Arbeiten

1.2.1 Datagram Transport Layer Security in Constrained Environments

Im Internet-Entwurf „Datagram Transport Layer Security in Constrained Environments“ [HB12] haben K. Hartke und O. Bergmann bereits einige Probleme aufgezeigt, die der Einsatz von DTLS in eingeschränkten

Umgebungen mit sich bringt und mögliche Lösungen vorgeschlagen.

Eines der Hauptprobleme ist hier die geringe Paketgröße in Netzen die IPv6 over Low power Wireless Personal Area Network (6LoWPAN) verwenden, da hier die Nutzdaten auf eine Länge von 127 Byte beschränkt sind. Insbesondere beim Aufbau der sicheren Verbindung (Handshake) müssen viele Daten ausgetauscht werden, was bei der Verwendung von DTLS die Paketgröße überschreiten würde. Lösen würde das Problem bspw. eine Nutzung der IP-Fragmentierung, was aber bei Verlust einzelner Pakete zu einem neuen Versand aller IP-Fragmente führen würde und somit umfangreichen DAtenverkehr erzeugt, der einen hohen Energieverbauch mit sich bringt. Ein weiterer Ansatz besteht darin, die Menge der Daten sowohl beim Verbindungsaufbau als auch bei der Datenübertragung durch Komprimierung der Headerdaten zu verringern wofür es unterschiedliche Vorschläge gibt. Bei Nutzung von CoAP wäre es auch möglich, den Verbindungsaufbau über CoAP zu realisieren. Dadurch ist die Transportsicherung gegeben und große Pakete könnten mit einer blockweisen Übertragung effizient übertragen werden, so dass bei Paketverlusten nur die verlorenen Pakete erneut übertragen werden müssten.

Beachtet werden müssen auch die Zeiten, nach denen ein Paket als verloren angesehen und erneut gesendet wird. Gerade beim Verbindungsaufbau kann es durch aufwendige Berechnungen, wie sie bspw. im Elliptic Curve Diffie-Hellman Schlüsselaustausch benötigt werden, zu einer erhöhten Antwortzeit kommen, was nicht zu einem erneuten Paketversand führen sollte.

Beim Verbindungsaufbau werden viele Daten ausgetauscht, was gerade in eingeschränkten Umgebungen einige Zeit dauern kann. Um die Zeit möglichst kurz zu halten, ist es wichtig die Anzahl der Kommunikationsvorgänge gering zu halten oder den Verbindungsaufbau schon durchzuführen bevor Anwendungsdaten ausgetauscht werden, damit diese dann sofort übertragen werden können.

Um Speicher zu sparen müssen auch die Anzahl der sicheren Verbindungen begrenzt werden und/oder Verbindungen nach einiger Zeit automatisch geschlossen werden um neue Verbindungen zu ermöglichen.

1.2.2 A Hitchhiker's Guide to the (Datagram) Transport Layer Security Protocol

Im Internet-Entwurf „A Hitchhiker's Guide to the (Datagram) Transport Layer Security Protocol" [TKK13] haben H. Tschofenig, S.S. Kumar und S. Keoh zunächst die Unterschiede von TLS 1.0, 1.1 und 1.2 erläutert und klargestellt, dass die Details beim Handshake von der Wahl des Ciphersuits abhängen. Anhand einiger Beispiele erläutern sie, dass es wichtig ist, sich der Position eines Gerätes in einer Verbindung bewusst zu sein. So kann ein Sensor mit beschränkten Ressourcen sowohl als Server als auch als Client realisiert werden wobei es auch auf die Anzahl der möglichen Verbindungen ankommt. Ein Sensor der als Client agiert wird mit großer Wahrscheinlichkeit immer nur einen Server kontaktieren um dort neue Sensordaten zu hinterlegen, während ein als Server realisierter Sensor durchaus auch Anfragen von mehreren Clienten erhalten kann. Je klarer die Position und die Umgebung des Sensors ist, desto weniger flexibel kann dieser Implementiert werden was den Aufwand und die Codegröße reduziert.

Im weiteren Verlauf gehen sie auf wichtige Design-Entscheidungen ein und Erläutern deren Bedeutung und mögliche Auswirkungen.

Kernstück des Entwurfs ist die Auswertung des Speicherverbrauchs, sowohl im Read-Only Memory (ROM) als auch im Random-Access Memory (RAM), und die Menge der Übertragenen Daten bei einem Handshake. Anhand eines modifizierten Prototypens zeigen sie dort auf, welche grundlegenden Teile von DTLS, ohne Berücksichtigung der Ciphersuit spezifischen Funktionen, wieviel Speicher verbrauchen und werten die Menge der übertragenen Daten in einem kompletten Handshake für die unterschiedlichen Protokollschichten aus. Des weiteren haben sie die Codegrößen von bspw. Hash-Funktionen und anderen für TLS notwendigen Berechnungen ausgewertet, wie sie in unterschiedlichen Ciphersuits verwendet werden.

Abschließend stellen sie fest, dass sie TLS/DTLS durchaus auf eingeschränkte Umgebungen zuscheiden lässt, wobei mehr Flexibilität aber zu einem größeren Programmcode führt.

1.3 Ziel

Ziel soll es sein DTLS so weit anzupassen, dass es sich in eingeschränkten Umgebungen, insbesondere auf einem Redbee Econotag [Red13] mit dem MC13224v Mikrocontroller [Fre13], nutzen lässt. Dabei soll der Funtkionsumfang von DTLS aber nicht eingegrenzt werden. Sämtliche im Standard definierten Möglichkeiten sollen weiterhin nutzbar sein und insbesondere durch Aushandlung eines Ciphersuits angewendet werden können.

Kapitel 2

Vorgehensweise

Im Vordergrund soll die Implementierung eines Sicherheitsprotokolls stehen, das sich an den Prinzipien von DTLS orientiert und einige der Vorschläge aus dem Internet-Entwurf von K. Hartke und O. Bergmann, und/oder eigene Ideen, realisiert. Dabei liegt ein besonderes Interesse darauf den Handshake über CoAP [She+12] zu realisieren und somit einige in DTLS eingefügte Konzepte überflüssig zu machen. Die Implementierung wird im Anschluss durch einen Vergleich mit DTLS evaluiert, wobei folgende Punkte eine Rolle spielen sollen: Volumen des generierten Traffics, Größe/Komplexität des Quellcodes und die vom Protokoll benötigte Speichermenge.

Die Implementierung besteht aus dem Clienten auf einem gängigen PC/Laptop, bei dem es keine speziellen Einschränkungen an Energie, Speicher oder Effizienz gibt, und aus dem Server, der für einen Redbee Econotag [Red13] mit dem MC13224v Mikrocontroller [Fre13] optimiert werden soll. Da der genannte Mikrocontroller die Verschlüsselung mit dem Advanced Encryption Standard (AES) im Counter (CTR)- und Cipher Block Chain (CBC)-Mode in Hardware unterstützt und die Rechenleistung sowie der Speicher beschränkt ist, soll nur ein Ciphersuit „TLS_PSK_ECDH_WITH_AES_128_CCM_8“, in Anlehnung an den RFC 6655 [MB12], realisiert werden. Dieses gibt einen Schlüsselaustausch mit Hilfe elliptischer Kurven vor, die mit kleineren Zahlen genau so sicher sind wie ein standard Diffie-Hellman-Schlüsselaustausch und sich somit effizient berechnen lassen. Zusätzlich wird auch ein Pre-Shared Key (PSK) verwendet, damit der Verbindungsaufbau nur den Clienten möglich ist, die über diesen Schlüssel verfügen. Die Verschlüsselung der Anwendungsdaten erfolgt dann im „Authenticated Encryption with Associated Data (AEAD)“ Modus [McG08] wobei sich hier „Counter with CBC-MAC (CCM)“ [WHF03] aufgrund der Hardwarevorraussetzungen am besten eignet. Dieser besteht aus einer Verschlüsselung der Daten durch AES im CTR-Modus während der dazugehörende Message Authentication Code (MAC) durch AES im CBC-Modus berechnet wird. Die Anzahl der möglichen sicheren Verbindungen soll beschränkt werden, um den Speicherverbrauch gering zu halten. Die dafür notwendigen verbindungsspezifischen Daten wie bspw. der Key-Block, sollen dabei im Flash-Speicher des Redbee Econotags abgelegt werden, so dass nur die für das aktuelle Paket benötigten Daten im RAM-Speicher befinden. Auf diese Weise lässt sich die Anzahl der möglichen sicheren Verbindungen trotz Beschränkung maximieren.

Bei der Evaluation soll die Datenmenge der Header-Daten sowohl beim Verbindungsaufbau als auch bei der

Übertragung von Anwendungsdaten mit einer reinen DTLS-Implementierung verglichen werden. Dabei werden nicht nur fehlerfreie Verbindungen betrachtet sondern auch Paketverluste mit einbezogen. Verglichen wird auch der notwendige Speicherbedarf bei Verbindungsaufbau und Übertragung der Anwendungsdaten. Ebenso soll die Größe/Komplexität des zugrunde liegenden Quellcodes bewertet werden.

Kapitel 3

TLS / DTLS

Das Sicherheitsprotokoll TLS [DR08] wird im Allgemeinen mit dem stromorientierten TCP verwendet. Client ??? Wurde durch TCP eine Verbindung hergestellt, können Daten von beliebieger Größe in jede Richtung übertragen werden. TCP wird dafür sorgen, dass der eingegebene Bytestrom vollständig und in der richtigen Reihenfolge auf der Gegenseite wieder ausgegeben wird. Um die TLS-bezogenen Daten nun zu kennzeichnen und voneinander abzugrenzen existiert das „Record Layer Protokoll“ dessen Header in Abbildung 3.1 dargestellt ist. Dort ist neben der Art des Inhalts und der Protokollversion auch die Länge enthalten, so dass aufeinanderfolgende Pakete im Datenstrom voneinander abgegrenzt werden können. Als Inhalt kommen 4 Sub-Protokolle in Frage. Währen das Application-Data-Protokoll für den Transport der Anwendungsdaten genutzt wird, kommt das Handshake-Protokoll für die Aushandlung der Sicherheitsparameter zum Einsatz. Über das Change-Cipher-Spec-Protokoll werden die zuletzt ausgehandelten Sicherheitsparameter aktiviert. Sollte es beim Handshake oder der Übertragung von Anwendungsdaten zu Fehlern kommen, werden diese mit Hilfe des Alert-Protokolls übertragen.

Da bei DTLS [RM12] im Allgemeinen das paketorientierte UDP verwendet wird, bei dem die Länge eines Paketinhalts bekannt ist, wirkt die Längenangabe zunächst überflüssig. Jedoch ist es insbesondere bei einem Handshake sinnvoll, mehrere DTLS-Pakete innerhalb eines UDP-Pakets zusammenzufassen, so dass auch hier wieder eine Längenangabe benötigt wird, um die Pakete voneinander abzugrenzen. Zusätzlich sind bei DTLS nun die Datenfelder für die Epoche und die Sequenz-Nummer hinzugekommen. Während diese beiden Werte bei TLS durch die gewährleistete Reihenfolge der Daten durch TCP implizit bekannt sind, müssen diese bei DTLS explizit angegeben werden, da UDP weder die Reihenfolge noch den Transport der Daten garantiert. Die Epoche wird bei einem erfolgreichen Handshake erhöht und ordnet so die dazugehörenden Daten den im Handshake ausgehandelten Sicherheitsparametern zu, während die Sequenznummer in jeder Epoche bei 0 beginnt und bei jedem Paketversand erhöht wird.

```
struct {
  ContentType type;
  ProtocolVersion version;
  uint16 epoch;                          // Nur bei DTLS
  uint48 sequence_number;                // Nur bei DTLS
  uint16 length;
  uint8  fragment[DTLS_Record.length];
} DTLS_Record;
```

Abbildung 3.1 Header des Record Layer Protokolls von TLS / DTLS

3.1 Handshake

Damit es überhaupt zu einer sicheren Verbindung kommen kann, müssen zunächst einige Sicherheitsparameter mit Hilfe des Handshake-Protokolls ausgehandelt werden. Der Header eines Handshake-Pakets setzt sich gemäß Abbildung 3.2 zusammen. Während es bei TLS ausreichend ist, den Typ, die Länge und die Daten selbst zu senden, wurden bei DTLS weitere Datenfelder ergänzt. message_seq dient zur Durchnummerierung der Handshake-Nachrichten um Paketverluste von UDP zu erkennen und eine mögliche falsche Reihenfolge der Pakete auszugleichen. Da UDP eine begrenzte Paketgröße hat und eine Fragmentierung der UDP-Pakete auf IP-Ebene vermieden werden soll, müssen Handshake-Nachrichten eventuell auf mehrere UDP-Pakete verteilt werden. Um dies zu ermöglichen wurden fragment_offset und fragment_length ergänzt. So können die Daten in mehrere Teile geteilt werden, während die Länge und die Position im Paket hinterlegt werden. length enthält nach wie vor die Gesamtlänge, so dass eine Fragmentierung jederzeit erkannt werden kann.

```
struct {
  HandshakeType msg_type;
  uint24 length;
  uint16 message_seq;                    // Nur bei DTLS
  uint24 fragment_offset;                // Nur bei DTLS
  uint24 fragment_length;                // Nur bei DTLS
  uint8  fragment[Handshake.length];
} Handshake;
```

Abbildung 3.2 Header des Handshake Protokolls von TLS / DTLS

Der bei einem Handshake entstehende Nachrichtenaustausch in vollständiger Form ist in Abbildung 3.3 aufgeführt. Die mit * markierten Pakete werden hier kurz erklärt, spielen aber im weiteren Verlauf keine Rolle, da die Authentifizierung durch den PSK realsiert werden soll und auf die Zertifikate verzichtet wird, um Ressourcen zu sparen.

Eingeleitet wird der Handshake mit einem ClientHello, in dem der Client seine Möglichkeiten bekannt gibt. Dazu gehören u.a. die unterstützten Protokollversionen, Ciphersuits und Kompressionsmethoden. Während der Server bei TSL nun direkt mit einem ServerHello und weiteren Handshake-Paketen antworten kann, lässt

```
                    Client           Server
                    ------           ------

       ClientHello  (seq=0) -------->

                            <-------- (seq=0)  HelloVerifyRequest    // Nur bei DTLS

       ClientHello  (seq=1) -------->                                // Nur bei DTLS
      (mit cookie)                                                   // Nur bei DTLS

                            <-------- (seq=1)  ServerHello
                            <-------- (seq=2) *Certificate
                            <-------- (seq=3)  ServerKeyExchange
                            <-------- (seq=4) *CertificateRequest
                            <-------- (seq=5)  ServerHelloDone

       Certificate* (seq=2) -------->
 ClientKeyExchange  (seq=3) -------->
 CertificateVerify* (seq=4) -------->
  ChangeCipherSpec          -------->
          Finished  (seq=5) -------->

                            <--------          ChangeCipherSpec
                            <-------- (seq=6)  Finished

 Application Data           <------->          Application Data
```

Abbildung 3.3 Nachrichtenaustausch während eines TLS / DTLS Handshakes

sich das bei DTLS so nicht realisieren. Da UDP kein verbindungsorientiertes Protokoll ist, können Pakete mit gefälschtem Absender versendet werden. Auf diese Art könnte ein Denial of Service (DoS) Angriff durchgeführt werden, in dem zahllose Pakete mit unterschiedlichen Absendern an den Server gesendet werden, welche alle ein ClientHello enthalten. Problematisch ist hierbei der Zustand der für jedes ClientHello im Server erzeugt wird. Neben dem Speicherverbrauch kann die Berechnung des ServerKeyExchange eine Menge Rechenleistung benötigen, so dass die Ressourcen des Servers schnell aufgebraucht sind. Um dies zu vermeiden und den Absender zu verifizieren wurde in DTLS ein Cookie ergänzt. Dieser wird aus dem ClientHello generiert und als Antwort an den Clienten gesendet. So kann der Server den Cookie bei einem erneuten ClientHello wieder berechnen und mit dem mitgelieferten vergleichen. Dadurch wird bei der ersten Anfrage ein Zustand vermieden und der Client verifiziert.

sofern kein Man-in-the Middle

Im ServerHello gibt der Server bekannt, welche der vom Client genannten Möglichkeiten ausgewählt wurden. Folgen können dann ein Zertifikat, Daten für einen Schlüsselaustausch sowie eine Anfrage für das Zertifikat des Clienten. Abschließend folgt ein ServerHelloDone um dem Clienten zu signalisieren, dass er wieder an der Reihe ist. Dieser sendet nun sein eigenes Zertifikat, falls vom Server angefordert. Es folgen Daten für den Schlüsselaustausch und Daten, die es dem Server ermöglichen das Zertifikat des Clienten zu überprüfen, falls dieses die Möglichkeit bietet Daten zu signieren. Damit sind zunächst alle Daten ausgetauscht, die für

die Aushandlung der Sicherheitsmechanismen notwendig sind.

Während die bisher genannten Handshake-Nachrichten mit den Sicherheitsparametern der aktuell gültigen Epoche versendet werden, folgt nun der Versand eines ChangeCipherSpec. Dieses Paket gehört formell nicht zum Handshake-Protokoll sondern bildet ein eigenes Protokoll, da hier die Epoche verändert wird. Ein ChangeCipherSpec Paket besteht ausschließlich aus einem 1 Byte langen Header mit dem Wert 1 und enthält keine weiteren Daten. Nach dem Versand des Pakets werden alle folgenden Pakete mit den Sicherheitsparametern der neuen Epoche versendet während erst der Empfang solch eines Pakets dazu führt, dass alle folgenden eingehenden Pakete mit Hilfe der neuen Sicherheitsparameter gelesen werden.

Schließlich wird noch eine Finished-Nachricht, die wieder zum Handshake-Protokoll gehört, ausgetauscht. Diese enthält einen Hash aller bisher ausgetauschten Daten und wird mit den Sicherheitsparametern der neuen Epoche verschlüsselt. So wird der Handshake verifiziert und die neuen Sicherheitsparemeter auf Korrektheit geprüft.

3.2 Alert

Wenn es während des Handshakes oder der Übertragung von Anwendungsdaten zu Fehlern kommt, werden diese mit Hilfe des Alert Protokolls übertragen. Der Header (siehe Abbildung 3.4) enthält neben dem AlertLevel, welches *warning* (1) oder *fatal* (2) sein kann, die Beschreibung des Fehlers. Während Fehler des Levels *fatal* zu einem unmittelbaren Verbindungsabbruch führen, sind Fehler des Levels *warning* zur Information der Gegenseite über mögliche Probleme gedacht. Das Alert-Protokoll unterscheidet sich bei TLS und DTLS nicht voneinander, da eine zuverlässige Übertragung nicht notwendig ist. Sollte aufgrund eines verloren gegangenen Alert-Pakets eine Anfrage wiederholt werden, wird erneut ein Alert-Paket generiert.

```
1 struct {
2   AlertLevel level;
3   AlertDescription description;
4 } Alert;
```

Abbildung 3.4 Header des Alert Protokolls von TLS / DTLS

Kapitel 4

Anpassungen

Während CoAP über Port 5683 betrieben wird, erfolgt Kommunikation über die mit DTLS gesicherte Variante nun über Port 5684 und trägt den Namen „coaps“ [Int13a]. In den folgenden Unterkapiteln wird beschrieben, welche Anpassungen am allgemeinen DTLS-Header möglich sind und wie der DTLS-Handshake über CoAP abgewickelt werden kann.

4.1 Header

Da die maximale Datenmenge eines Pakets im genutzen Datenübertragungsstandard IEEE 802.15.4 [LAN11] auf 127 Byte begrenzt ist würde der in DTLS definierte Header mit 13 Byte schon mehr als 10% des Datenvolumens ausmachen. Um das zu vermeiden wird die Stateless Header Compression aus dem Entwurf von K. Hartke und O. Bergmann [HB12, Kapitel 3] angewendet. Diese zeichnet sich durch eine verlustfreie Komprimierung aus, für die keine weiteren Informationen bereitgestellt werden müssen. Damit lässt sich der Header im besten Fall auf 2 Byte, wie in Abbildung 4.1 dargestellt, komprimieren.

```
 0                 1
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0| T | V |  E  |1 1 0|  S  | L |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Abbildung 4.1 Komprimierter Handshake-Header

Der RecordType (T) kann mit zwei Bit folgende 4 Zustände annehmen: *8-Bit-Feld* (0), *Alert* (1), *Handshake* (2) und *Anwendungsdaten* (3). Trotz Realisierung des Handshakes über CoAP ist diese Unterteilung notwendig, damit auch der DTLS-Layer über die Art des Inhalts informiert ist und speziell die direkt für ihn bestimmten Daten bearbeiten kann. Hierzu gehören die Daten des Alert-Protokolls, welche ohne CoAP

übertragen werden. Bei den Anwendungsdaten muss außerdem überprüft werden, dass diese nicht innerhalb der Epoche 0, ohne Sicherheitsparamater, versendet oder empfangen werden. Auf direkte Angabe von *ChangeCipherSpec* wurde verzichtet, da dies bei einem Handshake über CoAP nicht mehr notwendig ist (siehe Kapitel 4.2). Sollten weitere Unterptokolle notwendig sein, können diese innerhalb eines 1 Byte langen Typenfeldes an den Header gehangen werden, was durch den Wert 0 signalisiert wird. In diesem zusätzlichen Byte wird dann der im TLS/DTLS definierte Wert hinterlegt. So ist es auch möglich die 3 direkt definierten Werte unkomprimiert zu versenden. Die komprimierten Werte wurden so angeordnet, dass durch Addition von 20 die in TLS/DTLS definierten Werte ermittelt werden können.

Die Version (V) kann mit zwei Bit folgende 4 Zustände annehmen von denen 3 Benutzt werden: *DTLS 1.0* (0), *16-Bit-Feld* (1) und *DTLS 1.2* (2). DTLS 1.0 und DTLS 1.2 können hier direkt definiert werden, da DTLS 1.0 weit verbreitet und DTLS 1.2 die aktuellste Version ist. Auf DTLS 1.1 wurde verzichtet, da Implementierungen die über DTLS 1.0 hinaus gehen im Allgemeinen auch DTLS 1.2 unterstützen. Auch hier ist es möglich weitere Versionen an den Header anzuhängen, in dem V auf 1 gesetzt wird wobei hier mit 2 Byte das in TLS definierte Versionsformat zum Einsatz kommt.

Die Epoche (E) kann mit den Werten 0 bis 4 kann direkt angegeben werden. Da jede Kommunikation mit der Epoche 0 beginnt, und nach dem ersten Handshake in Epoche 1 fortgeführt wird, sind dies die am Häufigsten verwendeten Werte. Jeder weitere Handshake erhöht die Epoche um 1, so dass auch weitere Epochen möglich sind ohne den Header zu vergrößern. Sollten höhere Werte benötigt werden, lässt sich das mit den folgenden Zuständen realisieren: *8-Bit-Feld* (5), *16-Bit-Feld* (6) und *Implizit* (7). So können 8 oder 16 Bit lange Epochen an den Header gehängt werden, was den durch DTLS vorgebenen Bereich vollständig abdeckt. Alternativ kann durch den Wert 7 signalisiert werden, dass es sich bei der Epoche um die gleiche handelt, wie bei dem vorausgehenden DTLS-Paket innerhalb des gleichen UDP-Pakets.

Für die Sequenz-Nummer (S) sind mit drei Bit 8 Zustände möglich. Während mit den Werten 1 bi 6 die Länge in Byte der angehängten Sequenz-Nummer angegeben wird, kann durch den Wert 0 die Angabe unterbunden werden. Im Allgemeinen wird die Sequenz-Nummer in Verbindung mit der Epoche zur Berechnung des MACs herangezogen. Jedoch gibt es Ciphersuits die andere Mechanismen verwenden, so dass keine Sequenz-Nummer notwendig ist. Falls mehrere DTLS-Pakete innerhalb eines UDP-Pakets enthalten sind, kann die Sequenz-Nummer durch den Wert 7 auch relativ zum Vorgänger-Paket (+1) angegeben werden.

Schließlich folgt noch ein zwei Bit Wert für die Länge. Falls im UDP-Paket nur ein DTLS-Paket enthalten ist, kann hier der Wert 0 gesetzt werden, wodurch keine Länge angegeben wird. Diese ist durch die Länge des UDP-Pakets implizit bekannt. Mit den Werten 1 und 2 kann die Länge in Byte der angehängten Länge angegeben werden, während durch den Wert 3 das letzte DTLS-Paket im UDP-Paket gekennzeichnet wird, dessen Länge wieder implizit bekannt ist.

4.2 Handshake

Auch der Handshake orientiert sich am Entwurf von K. Hartke und O. Bergmann [HB12, Kapitel 4].

Wie in Kapitel 3 beschrieben, wurde der Header des DTLS-Handshake-Protokolls um eine Sequenz-Nummer und 2 Datenfelder für die Fragmentierung ergänzt. Diese sind zunächst notwendig, um die in UDP fehlende Zuverlässigkeit und begrenzte Paketgröße auszugleichen. Da der Handshake nun über CoAP realisiert wird, können diese Datenfelder jedoch wieder wegfallen, da CoAP über geeignete Mechanismen verfügt.

Durch eine Kennzeichnung aller CoAP-Anfragen während des Handshakes als „confirmable“ stellt CoAP sicher, dass alle Daten zuverlässig übertragen werden. Dies wird dadurch realisiert, dass auf jede Anfrage mit mindestens einem Acknowledgement (ACK)-Paket geantwortet wird. Bleibt dies aus, wird die Anfrage nach Ablauf einer Wartezeit wiederholt. Durch diese Zuverlässigkeit und den erhalt der Reihenfolge der Daten innerhalb eines CoAP-Pakets ist es somit möglich auf das „message_seq“ Datenfeld zu verzichten.

Um IP-Fragmentierung zu vermeiden, wird die Blockweise-Datenübertragung von CoAP verwendet [BS13]. Problematisch ist die Verwendung der IP-Fragmentierung, da bei Verlust eines einzelnen Fragments das ganze IP-Paket verworfen wird. So kommt es zu einer Wiederholung der CoAP-Anfrage und die Übertragung aller Fragmente wird wiederholt. Je nach Anzahl der Fragmente und der Paketverluste, kann es somit zu einer mehrfachen Übertragung, die sowohl Zeit als auch Energie benötigt. Um dies zu vermeiden werden die Daten schon durch CoAP in Fragmente unterteilt, die in einem einzelnen IP-Paket platz finden. Jedes Fragment wird dann durch ein eigenes CoAP-Paket übertragen welches als „confirmable“ gekennzeichnet ist. Geht ein Fragment verloren bleibt das ACK-Paket aus und die Übertragung nur dieses Fragments wird wiederholt. So bleibt die Menge der übertragenen Daten und damit der Energieverbrauch minimal und eine Übertragung der Daten ist auch bei einer hohen Rate an Paketverlusten möglich. Durch diesen Mechanismus kann auch auf die Datenfelder fragment_offset und fragment_length verzichtet werden.

Zu beachten ist jedoch die in CoAP genutze Blockgröße. Diese kann nur die Werte 2^x annehmen, wobei x im Bereich von 4 - 10 liegt. Unter Beachtung der maximalen Paketgröße von 127 Byte kommen hier somit nur die Blockgrößen 16, 32 und 64 in Frage. Es hat sich gezeigt, dass in der Testumgebung der Header eines 6LoWPAN-Paketes in das Sensornetz 48 Byte groß ist, während der Header eines 6LoWPAN-Paketes aus dem Sensornetz eine Größe von 40 Byte hat. Hinzu kommt jeweils noch der 8 Byte große UDP-Header, womit 71 bzw. 79 Byte für die CoAP-Anfrage bzw. -Antwort verbleiben. Der CoAP-Header ist minimal 4 Byte groß und eine Blockoption benötigt zusätzliche 3 Byte womit noch 64 bzw. 72 Byte für die Daten des Handshakes selbst bleiben. Da bei einer CoAP-Anfrage aber auch noch die Uniform Resource Identifier (URI) in den CoAP-Optionen hinzukommt, fällt die Blockgröße von 64 Byte für eine CoAP-Anfrage weg, so dass hier nur die Blockgrößen 16 und 32 zur Auswahl stehen. Bezieht man schließlich auch noch den DTLS-Header mit ein und berücksicht, dass ein Handschake auch innerhalb einer sicheren Verbindung durchgeführt werden kann, wobei dann noch ein 8 bis 16 Byte langer MAC hinzukommt, lassen sich auch CoAP-Antworten nur mit einer Blockgröße von 16 und 32 Byte übertragen.

Der vollständige Handshake über CoAP ist in Abbildung 4.2 zu sehen, wobei wieder die mit * markierten

Daten in dieser Arbeit keine Anwendung finden.

```
                    Client           Server
                    ------           ------

                POST /dtls ----1--->
           ClientHello

                             <---2---- 4.01 Unauthorized
                                        HelloVerifyRequest

                POST /dtls ----3--->
           ClientHello
          (mit cookie)

                             <---4---- 2.01 Created
                                        ServerHello (enthält Session X)
                                       *Certificate
                                        ServerKeyExchange
                                       *CertificateRequest
                                        ServerHelloDone

              POST /dtls/X ----5--->
           Certificate*
      ClientKeyExchange
      CertificateVerify*
       ChangeCipherSpec
              Finished

                             <---6---- 2.04 Changed
                                        ChangeCipherSpec
                                        Finished

           Application Data <-------> Application Data
```

Abbildung 4.2 Nachrichtenaustausch während eines TLS / DTLS Handshakes über CoAP

Für die Realisierung des Handshakes über CoAP dient die Ressource „/dtls“. Dieser URI wurde bewusst kurz gehalten, da er im Klartext in der CoAP-Anfrage eingefügt wird und so Daten und Energie spart. Während in einem gewöhnlichen DTLS-Handshake jedes einzelne DTLS-Paket einen vollständigen DTLS-Record-Header hat, ist dies hier nicht mehr notwendig, da mehrere DTLS-Pakete innerhalb eines CoAP-Pakets enthalten sind. Der DTLS-Record-Header erscheint somit einmalig vor jedem CoAP- oder Alert-Paket. Um die DTLS-Pakete innerhalb eines CoAP-Pakets voneinander abzugrenzen, dient nun der DTLSConten-Header (Abbildung reffig:com_content_header).

Dieser wurde abgeleitet vom Handshake-Header, wird jedoch nicht mehr so genannt, da in einem CoAP-Paket unterschiedliche DTLS-Inhalte enthalten sind. Neben Handshake- und ChangeCipherSpec-Paketen sind dort zusätzlich Alert-Pakete möglich. Während in den ersten sechs Bits der in TLS/DTLS definierte Wert für den Handshake-Typ hinterlegt werden kann, enthalten die letzten beiden Bits die Anzahl der dem

```
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|     T     | L |
+-+-+-+-+-+-+-+-+
```

Abbildung 4.3 Komprimierter Content-Header

Header folgenden Bytes der Länge, wobei der Wert 0 die Länge 0 direkt definiert. Neben den in TLS/DTLS definierten Handshake-Typen werden die folgenden beiden Typen definiert: *change_cipher_spec* (32) und *alert* (33).

Obwohl eine zuverlässige Übertragung von Benachrichtigungen gemäß DTLS nicht notwendig ist, macht es insbesondere bei einem Handshake sinn, diese innerhalb eines CoAP-Paketes zu versenden, falls es sich um die Antwort auf eine Anfrage handelt. Ausgehend von einem ClientHello, das auf der Server-Seite ein Problem auslöst, erwartet der Client vom Server eine CoAP-Antwort. Wird die Benachrichtigung darüber ohne CoAP versendet, muss der Client bei Erhalt der Benachrichtigung dafür sorgen, dass die Anfrage aus der darüber liegenden CoAP Schicht entfernt wird, damit diese nicht wiederholt wird. Bei Versand der Benachrichtigung über CoAP erledigt sich dies von selbst, da ja eine Antwort auf die Anfrage erhalten wurde. Der CoAP-Response-Code ist in diesem Fall „4.02 Bad Option“. Ein Beispiel dafür ist in Abbildung 4.4 zu sehen.

```
                    Client           Server
                    ------           ------

               POST /dtls -------->
          ClientHello

                              <-------- 4.02 Bad Option
                                           Alert(handshake_failure)
```

Abbildung 4.4 Nachrichtenaustausch während eines TLS / DTLS Handshakes über CoAP

Während der Record-Typ bei Anwendungsdaten eindeutig ist, muss dieser nun für einen Handshake und Benachrichtigungen definiert werden. Alert wird hier nur verwendet, falls es sich um eine Benachrichtigung direkt über UDP ohne CoAP handelt. Benachrichtigungen innerhalb von CoAP sind eindeutig durch den Content-Header gekennzeichnet und gehören immer zu einem Handshake womit hier auch der Record-Type Handshake verwendet wird. Handshake wird generell verwendet, wenn es sich um Handshake-Daten handelt. Dazu zählt hier nun auch ein enthaltenes ChangeCipherSpec-Paket. Während bei TLS/DTLS der Versand eines ChangeCipherSpec-Pakets zur anschließenden Änderungen der Sicherheitsparameter des Paketversands führt, kommmt es bei Empfang solch eines Pakets zur Änderungen der Sicherheitsparameter des Paketempfangs für alle folgenden Pakete. Diese Vorgehensweise ist hier nicht mehr notwendig. Die Epoche und somit die Sicherheitsparameter für den Paketempfang ergeben sich durch den DTLS-Header. Beachtet werden muss nur, wann eine alte Epoche für den Paketempfang für ungültig erklärt werden kann. Dieses ist

auf der Seite des Clienten nach erhalt der Nachricht Nr. 6 gemäß Abbildung 4.2 möglich da diese die letzte Nachricht der alten Epoche ist der Handshake erfolgreich abgeschlossen wurde. Der Server darf die alte Epoche für den Paketempfang jedoch erst nach Erhalt der ersten Anwendungsdaten in der neuen Epoche vernichten, da er nicht sicherstellen kann, dass Nachricht Nr. 6 den Clienten erreicht hat und diese somit Nachricht Nr. 5 wiederholen könnte. Genaus so verhält es sich für die Epoche und die dazugehörenden Sicherheitsparameter für den Paketversand. Hat der Server Nachricht Nr. 6 erhalten kann er alle weiteren Nachrichten mit den Sicherheitsparametern der neuen Epoche versenden und die alten löschen. Der Server weiß bei Erhalt der ersten Anwendungsdaten ebenfalls, dass Pakete innerhalb der alten Epoche nicht mehr versand werden und kann die dazu gehörenden Sicherheitsparameter vernichten.

Da somit gemäß Abbildung 4.2 das Finished-Paket innerhalb des CoAP-Pakets noch mit den alten Sicherheitsparametern verschlüsselt wird müssen hier zusätzliche maßnahmen Ergriffen werden um den Zweck des Pakets zu bewahren. Hier wird der Hash über alle im Handshake ausgetauschten Pakete nun zusätzlich mit den neuen Sicherheitsparametern verschlüsselt, wobei das das Finished-Paket durch das ChangeCipherSpec-Paket, welches einen Wechsel der Sicherheitsparameter kennzeichnet, eindeutig von den vorhergehenden Daten abgegrenzt wird. Das ChangeCipherSpec-Paket kennzeichnet somit nun immer einen Wechsel von der Epoche 0 ohne Verschlüsselung zur neuen Epoche mit Verschlüsselung. Je nach Ciphersuit wird dadurch die in TLS definierte Länge des Finished-Pakets von 12 vergrößert, da unter Umständen Zusatzinformationen wie Nonce und MAC hinzukommen.

Kapitel 5

Definition des Ciphersuits

Neben den grundlegenden Anpassungen im DTLS-Record-Layer und der Änderung des Handshakes ist es notwendig ein geeignetes Ciphersuit zu definieren. Dieses gibt vor, welche Mechanismen für die Authentisierung genutzt werden und auf Grundlage welcher Algorithmen diese durchgeführt wird. Auch kann ein Verfahren zum Schlüsselaustausch definiert werden. Für die Übertragung von Anwendungsdaten ist schließlich ein Verschlüsselungsverfahren festgelegt, das Vertraulichkeit und/oder Integrität sicherstellen soll. Das Ciphersuit beeinflusst somit wesentlich, welche und wieviel Daten während des Handshakes ausgetauscht werden und welche Berechnungen notwendig sind. Je nach Verfahren können diese sehr Umfangreich werden und sind somit insbesondere für eingeschränkte Umgebungen nicht immer geeignet.

Da der Mikrocontroller die Verschlüsselung mit AES-128 im CTR- und CBC-Mode in Hardware unterstützt, kommen zunächst die im RFC 6655 [MB12] aufgeführten Ciphersuits in Frage. Diese nutzen für die Verschlüsselung den CCM-Modus der durch die Hardware einfach und effizient realisiert werden kann. Während in Kapitel 3 einige Ciphersuits definiert sind, die zur Berechnung des Schlüssels das rechenaufwendige RSA-Verfahren benutzen, sind in Kapitel 4 einige Ciphersuits für AES-128 definiert, die einen PSK verwenden und von RFC 4279 [ET05] abgleitet wurden. Der einzige Unterschied besteht hier darin, dass für die Berechnung des MAC in RFC 4279 der Secure Hash Algorithm (SHA) verwendet wird.

1. TLS_PSK_WITH_AES_128_CCM
2. TLS_PSK_DHE_WITH_AES_128_CCM
3. TLS_PSK_WITH_AES_128_CCM_8
4. TLS_PSK_DHE_WITH_AES_128_CCM_8

Während 1 und 2 einen 16 Byte langen MAC benutzen, ist dieser bei 3 und 4 nur 8 Byte lang. Da der MAC an jedes verschlüsselte Datenpaket angehangen wird um die Integrität sicherzustellen, verkürzt sich dadurch die ohnehin geringe Menge an Anwendungsdaten (siehe Kapitel 4.2). Zwar ist es leichter einen 8 Byte langen MAC zu fälschen, da dieser anstatt 2^128 Werten, wie bei einem 16 Byte MAC, nur 2^64 Werte annehmen kann, aber dennoch fällt die Wahl hier zugunsten der größeren möglichen Datenmenge auf die 8 Byte lange Version womit Ciphersuit 1 und 2 nicht weiter betrachtet werden.

Im Unterschied zu Ciphersuit 3 wird bei Ciphersuit 4 nicht nur der PSK verwendet um den Schlüssel zu

berechnen. Zusätzlich wird ein Diffie-Hellman-Schlüsselaustausch durchgeführt und dessen Ergebnis mit dem PSK kombiniert. Der große Unterschied besteht darin, dass ein Angreifer mit Kenntnis des PSK ohne zusätzlichen Diffie-Hellman-Schlüsselaustausch jederzeit in der Lage ist den Schlüssel zu berechnen und die übertragenen Daten zu entschlüssel oder zu manipulieren, unabhängig davon ob er während des Handshakes gelauscht hat. Bei zusätzlicher Verwendung des Diffie-Hellman-Schlüsselaustauschs ist er nicht in der Lage den Schlüssel zu berechnen und die Daten zu entschlüssel. Problematisch bleibt nur ein Man-in-the-middle-Angriff während des Handshakes. Verhindert ein Angreifer die direkte Kommunikation und leitet den Handshake über sich, kann er mit dem PSK eine Verbindung zu beiden Parteien herstellen und den zukünftigen Datenverkehr mitlesen. Dies fällt erst dann auf, wenn der Angreifer verschwindet, da die beiden Parteien ohne ihn nicht direkt kommunizieren können. Die Verwendung von Ciphersuit 3 kommt somit nicht in Frage, womit Chipersuit 4 hier zunächst das Mittel der Wahl ist.

Problematisch ist jedoch, dass ein Diffie-Hellman-Schlüsselaustausch sehr rechenintensiv ist. Ein Versuch innerhab des GOBI-Projektes hat gezeigt, dass ein Diffie-Hellman-Schlüsselaustausch mit 128-Bit-Zahlen 2 mal ca. 30 Sekunden auf dem verwendeten Mikrocontroller benötigt. Um bei dem derzeitigen Stand der Technik eine ausreichende Sicherheit gewähleisten zu können, müssen jedoch minimal 1024-Bit-Zahlen verwendet werden, was außerhalb der Möglichkeiten des Mikrocontrollers liegt. Ein weiteres Manko ist auch die definierte Pseudo-Random-Funktion (PRF). Diese basiert auf Keyed-Hashing for Message Authentication (HMAC) mit SHA2 was die größe des Programms relevant erhöht. Laut dem Internet-Entwurf „A Hitchhiker's Guide to the (Datagram) Transport Layer Security Protocol“ [TKK13] von H. Tschofenig, S.S. Kumar und S. Keoh werden 2.928 Byte für HMAC und 2.432 Byte für SHA benötigt. SHA2 ist hier leider nicht mit aufgeführt. Zu Beachten ist bei diesen Angaben jedoch, dass es sich hier um 64-Bit-Code handelt. Da der in dieser Arbeit benutzte Mikrocontroller jedoch mit 16-Bit-Code betrieben wird, ist hier eher eine Größe von ca 1200 - 1500 Byte zu berücksichtigen, die nur für die PRF benötigt wird. Um diese Probleme zu lösen bzw. die benötigte Codegröße und Rechenleistung zu reduzieren wird nun ein neues Ciphersuit definiert.

5.1 TLS_PSK_ECDH_WITH_AES_128_CCM_8

In Ahnlehnung an das Ciphersuit 4 soll nun zunächst die Effizienz des öffentlichen Schlüsselaustauschs verbessert werden. Um das zu realisieren wird hier nun die Elliptic Curve Cryptography (ECC) gemäß RFC 4492 [Bla+06] für einen Diffie-Hellman-Schlüsselaustausch verwendet werden. Bei der Verwendung von 256-Bit-Zahlen ist hier eine höhere Sicherheit gegeben als bei der Verwendung von 2048-Bit-Zahlen in einem gewöhnlichen Diffie-Hellman-Schlüsselaustausch. ECC und AES-CCM wurden zwar Internet-Entwurf „AES-CCM ECC Cipher Suites for TLS“ [McG+11] schon kombiniert, jedoch werden hier Zertifikate anstatt eines PSK verwendet, die hier nicht genutzt werden sollen.

Durch die Verwendung von ECC bekommt das Ciphersuit nun seinen Namen:

- TLS_PSK_ECDH_WITH_AES_128_CCM_8

Da dieses kein offizielles Ciphersuit gemäß den „Transport Layer Security (TLS) Parameters“ [Int13b] ist, wird hier die Nummer {0xFF,0x01} benutzt, da diese für die private Nutzung reserviert ist.

Alle weiteren für einen Diffie-Hellman-Schlüsselaustausch unter Verwendung von ECC sind Teil der Aushandlung im Handshake und müssen hier somit nicht weiter definiert werden.

Die Nonce setzt sich gemäß dem RFC 5116 [McG08] für die Umsetzung von AEAD-Algorithmen zusammen. Es wird hier eine 12 Byte lange Nonce verwendet, aus einem Initialisierungsvektor, der Epoche und der Sequenznummer zusammensetzt (siehe Abbildung 5.1). Während die Epoche und die Sequenznummer

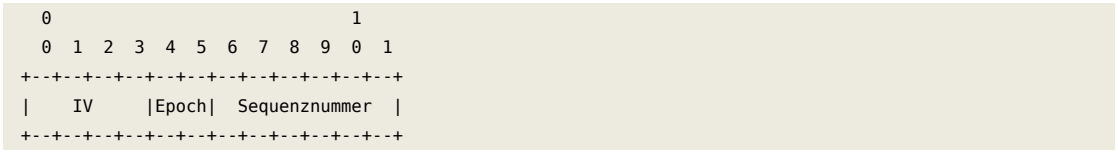

Abbildung 5.1 Nonce für AES-CCM

im DTLS-Header enthalten sind und in der selben Ordnung (Network Byte Order == Most Signifikant Byte first) in der Nonce hinterlegt werden, wird der Initialisierungsvektor nicht mit übertragen und ist implizit bekannt durch Erzeugung des Schlüsselblocks innerhalb des Handshakes. Dieser wird gemäß RFC 5246 [DR08] Kapitel 6.3 durch die PRF erzeugt und muss somit 40 Byte lang sein, welche wie folgt aufgeteilt sind:

- 0 Byte: client_write_MAC_key
- 0 Byte: server_write_MAC_key
- 16 Byte: client_write_key
- 16 Byte: server_write_key
- 4 Byte: client_write_IV
- 4 Byte: server_write_IV

Separate Schlüssel für die Verschlüsselung der Daten und der Erzeugung des MAC sind hier nicht mehr notwendig, da bei der Verwendung von AES-CCM mit einem Schlüssel beide Dinge erledigt werden.

In RFC 5116 [McG08] wird für die Umsetzung von AES-CCM auf eine Veröffentlichung des National Institute of Standards and Technology (NIST) mit der Nummer 800-38C [Nat04] verwiesen. Diese beschreibt das Verfahren in gleicher weise wie RFC 3610 [WHF03], wobei dieser bei den weiteren Erläuterungen nun Anwendungen finden soll. Für die Umsetzung sind 2 Parameter notwendig. Zum einen muss die Länge des MAC (M) definiert werden, welche in diesem Fall schon durch den Wert 8 festgelegt ist. Zum anderen muss die Größe des Längenfeldes (L) definiert werden, welches die Länge der zu verschlüsselnden Daten enthält und somit die Länge der Daten begrenzt. Da sich die Länge der Nonce durch 15 - L ergibt, und die Länge der Nonce bereits auf 12 festgelegt ist, ergibt sich hier der Wert für L von 3. Damit ist die Länge beschränkt auf 3 MiB, was aber mehr als ausreicht, da in dem betrachteten Umfeld ein Paket maximal 127 Byte groß sein kann, und die zu verschlüsselnde Datenmenge aufgrund der abzuziehenden Header noch weit darunter liegt.

Um auf HMAC und SHA2 verzichten zu können wird nun noch die PRF für dieses Ciphersuit definiert. Anwendung findet diese in 3 Fällen zur Berechnung folgender Werte:

master_secret
: PRF(pre_master_secret, „master secret“, client_random + server_random)

key_block
: PRF(master_secret, „key expansion“, server_random + client_random)

finished
: PRF(master_secret, finished_label, Hash(handshake_messages))
wobei finished_label = „client finished“ oder „server finished“

Grundlage zur Berechnung soll ein Cipher-based Message Authentication Code (CMAC) auf Basis von AES sein, der in RFC 4493 [Son+06] definiert ist, wobei der PSK direkt als Schlüssel genutzt wird. Nach Vorbild von RFC 5246 [DR08] wird die PRF nun gemäß Abbildung 5.2 definiert, wobei + die Konkatenation 2er Zeichenketten bedeutet.

```
1 PRF(secret, label, seed) = P_hash(secret + label + seed)
2
3 P_hash(seed) = CMAC(A(1) + seed) +
4                CMAC(A(2) + seed) +
5                CMAC(A(3) + seed) + ...
6 A(0) = seed
7 A(i) = CMAC(A(i-1))
8
9 CMAC(data) = AES-CMAC(psk, data)
```

Abbildung 5.2 Definition der Pseudo-Random-Funktion

Um den Hash für die Berechnung der Finished-Nachricht zu ermitteln, wird nun ebenfalls der definierte CMAC benutzt. Um die Berechnung zu vereinfachen wird der Hash entgegen dem Vorschlag von K. Hartke und O. Bergmann [HB12] aus den Handshake-Nachrichten ermittelt, wie sie über das Netz versendet wurden. Sollte also eine Stateless Header Compression wie in Kapitel 4.1 verwendet werden, dann wird der Hash auf Grundlage der komprimierten Nachrichten berechnet. So können die ein- und ausgehenden Nachrichten direkt für die Berechnung des Hashs gespeichert werden, ohne weitere Berechnungen vornehmen zu müssen.

Kapitel 6

Praktische Umsetzung

In den folgenden Abschnitten werden wichtige Merkmale der praktischen Umsetzung erläutert und einige Details erklärt, ohne eine umfangreiche Dokumentation des Quellcodes zu erstellen. Die Dokumentation des Quellcodes erfolgt für öffentliche Funktionen in den Header-Dateien im Stil von Doxygen [Hee13].

Das im vorigen Kapitel definierte Ciphersuit hängt grundlegend vom PSK des Endgeräts (Server) ab. Jeder, der diesen kennt, ist in der Lage, während eines Handshakes einen Man-in-the-middle-Angriff durchzuführen oder Werte der PRF zu Berechnen, da diese auf dem PSK basiert. Jedes Endgerät wird bei Herstellung mit einem eigenen PSK ausgerüstet. Dies wird durch ein Programm namens „Blaster“ realisiert, das im Bachelor-Projekt GOBI entstanden ist und für die Verwendung in dieser Arbeit angepasst wurde. Während in GOBI eine Persönliche Identifikationsnummer (PIN) generiert wurde, die nach Erstellung einer sicheren Verbindung zur Authentifizierung des Besitzers des Endgeräts benutzt wurde, wird hier nun ein PSK generiert. Blaster kommt zum Einsatz, nachdem der Quellcode des Endgeräts kompiliert wurde und erweitert die Binärdatei um Daten, die nach dem, maximal ~96 KiB großen, Programmcode folgen. Diese, maximal 28 KiB, werden nicht mit in den RAM-Speicher kopiert und können zur Ablage von Daten genutzt werden, die auch bei einem Batterie-Wechsel erhalten bleiben sollen. Neben dem PSK wird auch ein Universally Unique Identifier (UUID) generiert um das Endgerät eindeutig zu identifizieren. Da diese Daten für den Aufbau der DTLS-Verbindung genutzt werden, müssen diese einem Endgerät beigelegt werden, was durch einen Aufkleber auf der Verpackung realisiert werden könnte. Um einem Benutzer das Einbinden neuer Endgeräte möglichst einfach zu machen, wurde Blaster so erweitert, dass bei Ausführung auch ein QR-Code generiert wird. So kann der QR-Code frühzeitig in einem DTLS-Clienten hinterlegt werden, so dass die Daten bei einem Verbindungsaufbau direkt verfügbar sind. Dieses System hat den Nachteil, dass der PSK unter Umständen mindestens einem Vorbesitzer des Endgeräts bekannt ist. Dieser soll aber nach Veräußerung eines Endgeräts keinen Zugriff mehr darauf bekommen. Um dem Vorzubeugen ist der dem Endgerät beiliegende PSK nur für einen Verbindungsaufbau gültigt. Ist dieser erfolgreich abgeschlossen, wird automatisch ein neuer PSK generiert und bei einem weiteren Verbindungsaufbau benutzt. Möchte der Besitzer eine weitere Verbindung zum Endgerät aufbauen, kann er den neuen PSK über die vorhandene sichere Verbindung abrufen und nutzen, wobei dann wieder ein neuer PSK generiert wird. Um ein Endgerät zu veräußern, kann ein Reset-Knopf gedrückt werden, welcher das Endgerät auf den Werkszustand zurücksetzt und so den ur-

sprünglichen PSK wieder aktiviert.

6.1 Server

Der Server wird auf einem Redbee Econotag [Red13] realisiert. Der darauf enthaltene Mikrocontroller MC13224v [Fre13] enthält, neben dem IEEE 802.15.4 Funkstandard und einer AES Hardware-Engine, 128 KiB Flash-Speicher und 96 KiB RAM-Speicher. Bei Inbetriebnahme wird das im Flash-Speicher vorliegende Programm vollständig in den RAM-Speicher kopiert und dort ausgeführt, wodurch sich eine maximale Programmgröße von 96 KiB ergibt. Die zusätzlichen 32 KiB Flash-Speicher können somit für die Ablage von Daten genutzt werden, die auch nach einer Stromunterbrechung, oder einem Neustart des Geräts, erhalten bleiben sollen. Zu berücksichtigen ist jedoch auch noch, dass der letzte 4 KiB große Block schon für den Redbee Econotag selbst reserviert ist.

Betrieben wird der Server mit SmartAppContiki [Kov13], das auf Contiki [Con13] basiert, und eine Implementierung von CoAP, in der Entwurfsversion 13 [She+12], enthält. In der Standardkonfiguration benötigt SmartAppContiki, mit einer definierten CoAP-Ressource, die ein „Hallo Welt!“ zurückgibt, ~84 KiB. Diese Daten teilen sich gemäß Abbildung 6.1 auf. Um den benötigten Speicher zu optimieren wurde die Größe des „Sys Stack“ und des „Heap“ in der Konfigurationsdatei „contiki/cpu/mc1322x/mc1322x.lds“ angepasst.

Beschreibung	**Standard**	**Angepasst**
Programm	59248 Byte	59248 Byte
Irq Stack	256 Byte	256 Byte
Fiq Stack	256 Byte	256 Byte
Svc Stack	256 Byte	256 Byte
Abt Stack	16 Byte	16 Byte
Und Stack	16 Byte	16 Byte
Sys Stack	1024 Byte	2048 Byte
Datensegment	20872 Byte	20872 Byte
Heap	4096 Byte	16 Byte
Gesamt	86040 Byte	82984 Byte
	≈ 84 KiB	≈ 81 KiB

Abbildung 6.1 Speicheraufteilung von SmartAppContiki

Das wurde möglich durch Verwendung der in Contiki eingebauten Beobachtungswerkzeuge. Durch Definieren von periodischen Ausgaben der benutzen Heap sowie Sys Stack Größe, in „contiki/platform/redbee-econotag/contiki-mc1322x-main.c“, können die Auslastungen beobachtet werden. Um diesen Prozess effizienter zu gestalten, wird nur die Initialisierung durchgeführt und die periodischen Ausgaben deaktiviert. In „server/server.c“ lässt sich nun, durch Aktivieren des Debug-Modus, ein Code einbinden, der auf Knopfdruck sowohl die Speicheraufteilung als auch die bisher genutzten Bytes des Sys Stack und Heap ausgibt. Dadurch lässt sich erkennen, dass der Heap garnicht benutzt wird, und somit unnötig Speicher belegt. Da

insbesondere während des Handshakes, unter anderem aufgrund der Berechnung von elliptischen Kurven, viele Daten zwischengespeichert werden müssen, wird ersichtlich, dass ein Sys Stack von 1024 Byte nicht ausreicht, eine Größe von 2048 Byte jedoch optimal ist. Durch diese Anpassungen wurde der, für SmartAppContiki benötigte, Speicher von ~84 KiB auf ~81 KiB reduziert (siehe Abbildung 6.1). Somit stehen für die Umsetzung von DTLS ~15 KiB zur Verfügung, wobei auch berücksichtig werden muss, dass noch die Funktionen des Geräts selbst implementiert werden müssen.

Bei der Benutzung der, in SmartAppContiki enthaltenen, CoAP 13 Implementierung, hat sich herausgestellt, dass die Unterstützung für die CoAP-Option „Block 1“ fehlt. Diese Option wird von einem Clienten benutzt, um größere Datenmengen, in einer CoAP-Anfrage, in Blöcke zu unterteilen, damit es nicht zu einer Fragmentierung auf IP-Ebene kommt. Da diese Funktion für den DTLS-Handshake notwendig ist, wurde sie für Datenmengen bis 128 Byte nachgerüstet, wobei dieser Wert in der Datei „contiki/apps/er-coap-13/er-coap-13.h“ durch Änderung von „COAP_BLOCK1_BUFFER_SIZE“ eingestellt werden kann. Übertragen werden mit Hilfe der Option ein ClientHello oder ClientKeyExchange + ChangeCipherSpec + Finished, wobei die Länge von letzterem mit insgesamt 112 Byte (87 + 3 + 22), durch die einzig definierte Ciphersuit, fest ist. Anders ist dies bei dem ClientHello, welches durch eine Vielzahl, vom Client beherrschter Ciphersuits, sehr viel größer als 128 Byte werden kann. Hier erfolgt dann jedoch eine CoAP-Fehlermeldung „5.01 NOT IMPLEMENTED“ mit einem Hinweis auf die begrenzte Maximalgröße, so dass ein Client sein Angebot an Ciphersuits reduzieren kann, um der maximalen Größe gerecht zu werden.

Während bisher jede CoAP-Anfrage in der CoAP 13 Implementierung zu einem Aufruf des entsprechenden Ressource-Handlers geführt hat, verhält sich dies bei einer CoAP-Anfrage mit der Block 1-Option anders. Hier werden die Daten zunächst in einem Buffer gesammelt und ein Aufruf des Ressource-Handlers vermieden, wobei der Empfang jedes Blocks bestätigt wird. Erst bei Eintreffen des letzten Blocks, wenn die Anfrage vollständig ist, wird der Ressource-Handler aufgerufen, um die Anfrage zu bearbeiten und eine Antwort zu generieren. Um an die Daten der Anfrage zu gelangen, dient die Methode „coap_get_payload“. Diese wurde so angepasst, dass sie bei einer Anfrage ohne Block 1-Option wie gewohnt auf die Daten des aktuellen Pakets zeigt, bei einer Anfrage mit Block 1-Option jedoch auf den Buffer zeigt. In diesem Moment wird der Buffer als gelesen markiert, so dass ein erneuter Aufruf von „coap_get_payload“ wieder auf die Daten des aktuellen Pakets zeigt.

Während bisher allgemeine Anpassungen von SmartAppContiki bzw. dem darin enthaltenen CoAP 13 beschrieben wurden, folgt in den nächsten 3 Abschnitten die Erläuterung von 3 implementierten Contiki-Apps, welche für die Realisierung von DTLS notwendig sind. Die Implementierung von DTLS wird dann im 4. Abschnitt erläutert, wonach abschließend noch eine Update-Funktion erläutert wird, die für DTLS nicht notwendig ist, aber dessen Umfeld berücksichtigt.

6.1.1 Contiki-App: „flash-store“

Für eine Nutzung des erweiterten Flash-Speichers wird die App „flash-store“ verwendet. Als Basis für die Implementierung dient Code aus dem Bachelor-Projekt GOBI. Dieser war jedoch noch nicht als Contiki-App organisiert sondern direkt mit in den Code eingebunden. Auch ist die Aufteilung der 4 KiB großen Flash-Speicher-Blöcke eine andere. Diese Aufteilung ist in Abbildung 6.2 zu sehen. Während oben die 8 Speicherblöcke mit ihren Adressen aufgeführt sind, werden darunter die Aufteilungen für unterschiedliche Zwecke angegeben, wobei dort sowohl die GOBI-Aufteilung als auch die neue Aufteilung aufgeführt sind.

$0x18000$ – $0x18FFF$	$0x19000$ – $0x19FFF$	$0x1A000$ – $0x1AFFF$	$0x1B000$ – $0x1BFFF$	$0x1C000$ – $0x1CFFF$	$0x1D000$ – $0x1DFFF$	$0x1E000$ – $0x1EFFF$	$0x1F000$ – $0x1FFFF$

Aufteilung innerhalb des Bachelor-Projekts GOBI:

RO 1	RW 1	RW 2	RO 2	SR
	$0x0000 - 0x0FFF$	$0x1000 - 0x1FFF$	← virtuelle Speicheradressen	

Neue Aufteilung für DTLS:

RW 1	RW 2	RAD	RO	SR
$0x0000 - 0x0FFF$	$0x1000 - 0x1FFF$	← virtuelle Speicheradressen		

Legende: RW = Read-Write, RAD = Read-Append-Delet, RO = Read-Only, SR = System-Reserved

Abbildung 6.2 Aufteilung des erweiterten Flash-Speichers

Geändert wurde zunächst die Position der beiden RW-Blöcke. Diese ermöglichen das Schreiben von Daten, ohne die Eigenschaften des Flash-Speichers berücksichtigen zu müssen. Dieser kann nur beschrieben werden, falls die betroffene Position vorher einmal gelöscht wurde, was sich aber nur in Blöcken a 4 KiB realisieren lässt. Um Datenverluste zu vermeiden, werden jeweils zwei 4 KiB große Blöcke benutzt, um einen 4 KiB großen Speicher zu realisieren, der sich durch virtuelle Adressen ansprechen lässt, welche ebenfalls in Abbildung 6.2 aufgeführt sind. Die Daten sind immer nur in einem Block gespeichert, während der andere Block gelöscht ist. Kommt es zu einem Schreibvorgang, wird der Datenblock in den leeren Block kopiert, wobei die gewünschten Änderungen realisiert werden. Die Position der beiden RW-Blöcke befindet sich nun am Anfang, da sich so die Adressen, der jeweils zusammengehörenden Blöcke, genau um ein Bit unterscheiden. Das vereinfacht die Berechnung der Quell- und Ziel-Adresse erheblich, so dass durch Optimierung des Quellcodes 70 Byte an Programmgröße eingespart werden.

Problemtisch ist jedoch die Dauer und der Energieverbrauch bei einem Schreibzugriff dieser Art. Um eine effizientere Ablage von Daten zu ermöglichen folgt nach den beiden RW-Blöcken anstatt des RO-Blocks nun

ein RAD-Block. Dieser ist vergleichbar mit einem Stack ohne Push- und Pop-Funktion. Für die Initialisierung wird der komplette Block gelöscht. Daten können nun so lange eingefügt werden, bis der Block voll ist. Wieviel Daten gerade enthalten sind, wird dabei in einer globalen Variablen im RAM-Speicher gespeichert. Der Lesezugriff kann dabei beliebig erfolgen.

Gleich geblieben ist die Position des RO-Blocks. Dort können im Vorfeld, durch das bereits erwähnte Programm „Blaster", Daten abgelegt werden, welche zur Laufzeit ausgelesen werden können. Dies spart Programmgröße, da diese „Konstanten" nicht im Datensegment des Programms enthalten sind.

Abschließend folgt noch ein Block der für den Redbee Econotag selbst reserviert ist, und somit nicht genutzt werden kann.

6.1.2 Contiki-App: „time"

Im Gegensatz zu herkömmlichen Desktop-Rechnern oder Servern verfügt der Redbee Econotag über keine innere Uhr bezüglich der Realzeit. Angeboten wird vom MC13224v das Register „MACA_CLK" welches mit einem Takt von 250 KHz erhöht wird. Dieses läuft, bedingt durch seine Breite von 32 Bit, jedoch alle 4,77 Stunden über, so dass ohne weitere Eingriffe keine direkte Berechnung der Zeit möglich ist. Ähnlich verhält es sich mit dem Register „CRM_RTC_COUNT" welches im Takt von „CRM_RTC_TIMEOUT" Hz erhöht wird. Dieser Wert wird von Contiki eingestellt und liegt bei ~20 KHz. Dadurch erfolgt hier ein Überlauf nach ungefähr 60 Stunden. Neben dem Überlauf haben beide Quellen das Problem, dass die Register bei Einschalten des Econotags bei 0 anfangen, und die Werte somit keinerlei Bezug zur Realzeit haben.

Contiki löst einen Teil der genannten Probleme und stellt die Funktion „clock_seconds" zur Verfügung. Diese kümmert sich um den Überlauf und berechnet laufend die, seit dem Einschalten des Econotags, vergangenen Sekunden. Diese werden in einer 32-bit-Variablen gespeichert, wodurch ein Überlauf erst nach ~136 Jahren vorkommen kann.

Um den Bezug zur Realzeit herzustellen, wird die aktuelle Unixzeit vom Blaster generiert und im RO-Teil des Flash-Speichers hinterlegt. Diese Zeit spiegelt somit den Herstellungszeitpunkt wieder. Wird die aktuelle Zeit benötigt, kann diese durch Addition der von Contiki ermittelten Sekunden und der Unixzeit berechnet werden. Das funktioniert natürlich nur so lange, wie der Redbee Econotag nach dem Flashen ununterbrochen mit Strom versorgt wird. Um eine Korrektur zu ermöglichen, bietet die App eine Methode an, um die aktuelle Uhrzeit zu setzen. Diese wird mit der alten Zeit vergleichen um einen Korrekturwert zu ermitteln der in einer globalen Variablen hinterlegt wird. Hier macht es keinen Sinn diesen im Flash-Speicher abzulegen, da er nach einem Batteriewechsel direkt wieder veraltert wäre.

6.1.3 Contiki-App: „ecc“

Für die Berechnung von elliptischen Kurven wurde im Bachelor-Projekt GOBI von Jens Trillmann ein C-Programm implementiert. Dieses basiert auf einer Implementierung für einen 8-bit Mikrocontroller [Coco9], wurde jedoch für 32-bit-Prozessoren optimiert. Getestet und benutzt wird diese Implementierung bisher nur auf Desktop-Rechnern, wobei hier die Ausführung der Berechnungen in nicht wahrnehmbarer Zeit erledigt wird.

Um die Berechnungen auch auf dem Redbee Econotag durchzuführen, wird die Implementierung in eine eigene Contiki-App übernommen. Da der MC13224v-Mikrocontroller ebenfalls 32-bit-Berechnungen durchführt, ist dies zunächst direkt möglich. Im Gegensatz zu Prozessoren in Desktop-Systemen arbeitet der Mikrocontroller jedoch mit einer wesentlich geringen Taktfrequenz, so dass sich die benötigte Rechenzeit für eine Multiplikation auf elliptischen Kurven auf 13 Sekunden beläuft. In Zusammenarbeit mit Jens Trillmann sind deshalb zunächst die drei Grundfunktionen „Addition“, „Subtraktion“ und „Right-Shift“ für große Zahlen in Assembler realisiert worden, um die Berechnung schneller zu machen. Weitere Optimierungen sollen in der Bachelorarbeit von Jens Trillmann folgen.

Auf Basis des „ARM GCC Inline Assembler Cookbook“ [KK13] sind für die drei Grundfunktionen einige Varianten entstanden. Welche davon jeweils genutzt wird, lässt sich in den einzelnen Quellcode-Dateien einstellen. Generell bietet sich eine Umsetzung in Assembler an, da sich das sogenannte „Carry-Bit“ nutzen lässt. In diesem wird bei einer Rechenoperation ein möglicher Überlauf gespeichert. Für die gängigen Rechenoperationen gibt es zwei unterschiedliche Befehle, wobei nur bei einem das Carry-Bit genutzt wird. Dieses Potenzial zu nutzen, hat sich jedoch als schwierig herausgestellt, da Contiki das Thumb-Instruktion-Set des MC13224v nutzt. Im Gegensatz zum ARM-Instruktion-Set, dass 32-bit-Operationen nutzt, sind es im Thumb-Instruktion-Set nur 16 Bit. Jede Thumb-Instruktion wird bei Ausführung automatisch in die entsprechende ARM-Instruktion umgewandelt und ausgeführt. Durch die begrenzte Größe stehen jedoch nicht alle ARM-Instruktionen zur Verfügung und die Anzahl der nutzbaren Register ist auf 8 reduziert. Der Vorteil liegt jedoch in der geringen Programmgröße, so dass Contiki überhaupt erst auf dem MC13224v betrieben werden kann.

Alle drei Grundfunktionen sind zunächst ohne Einschränkung, der auch in C implementierten Funktionalität, umgesetzt. Insbesondere sind somit die Längen der Ein- und Ausgabewerte variabel, was sich nur mit einer Schleife realisieren lässt. Eine Schleife bedeutet jedoch auch, dass ein Zähler erhöht und verglichen werden muss. Da der Block mit dem Carry-Bit im Thumb-Instruktion-Set durch alle Operationen aktualisiert wird, geht das Carry-Bit der Hauptoperation vom einen zum nächsten Schleifendurchlauf verloren, muss manuell zwischengespeichert, und bei Bedarf berücksichtigt werden. Ein Sichern und Wiederherstellen des Blocks mit dem Carry-Bit ist nur im ARM-Instruktion-Set möglich. Die Optimierung besteht bei dieser Umsetzung somit nur darin, dass es einfach möglich ist, einen Überlauf zu erkennen. Während dies bei der Addition und Subtraktion jeweils ~20 Byte Programmgröße einspart, bringt es bei Right-Shift keinen Größenvorteil. Jedoch ist die Berechnung aufgrund der eingesparten Vergleiche bei allen Operationen schneller.

Da im Thumb-Instruktion-Set für einen Right-Shift keine Funktion zur Verfügung steht, die das Carry-Bit direkt benutzt, ist hier keine weitere Optimierung möglich. Für die Addition und Subtraktion sind weitere Varianten verfügbar. Da die Subtraktion ausschließlich für die Berechnung von 256-bit-Zahlen benutzt wird, was acht 32-bit-Blöcken entspricht, ist es möglich, die acht Subtraktionen direkt hintereinander auszuführen, so dass das Carry-Bit ohne weitere Eingriffe direkt berücksichtigt wird. Die Programmgröße nimmt dabei, im Vergleich zum C-Code, um ~12 Byte zu, wobei die Berechnungsgeschwindigkeit jedoch wesentlich zunimmt. Anders verhält es sich bei der Addition, da Werte unterschiedlicher Größe addiert werden. Notwendig sind hier 32, 64, 128, 256 und 512 Bit. Für jede dieser Größen ist nun ein eigener Additionsblock vorhanden. Bei Aufruf der Funktion wird die Größe überprüft und der richtige Block ausgewählt. Dies bietet eine maximale Geschwindigkeit, erhöht jedoch die Größe des Programms um ~212 Byte. Redzuieren lässt sich dieses noch, durch die Realisierung der 512-bit-Addition, durch zwei bis drei 256-Bit-Additionen, wobei hier, das dabei entstehende Carry-Bit, manuell berücksichtigt werden muss. Im Vergleich zum C-Code ist diese Variante 168 Byte größer.

Um weitere 96 Byte einzusparen, sind 3 benötigte Konstanten im Flash-Speicher hinterlegt. Dazu gehören die X- und Y- Koordinate des Basis-Punkts, der für den Diffie-Hellman-Schlüsselaustausch verwendet wird, und die Ordnung der verwendeten elliptischen Kurve. Die Ordnung wird nur verwendet, um zu überprüfen, ob der zufällig generierte private Schlüssel sich für die Benutzung eignet. Bei Bedarf werden die Werte aus dem Flash-Speicher geladen und nur so lange im Stack abgelegt, wie sie benötigt werden.

Weitere Optimierungen bezüglich der Programmgröße und Berechnungsgeschwindigkeit sollen in der Bachelorarbeit von Jens Trillmann folgen.

6.1.4 Contiki-App: „er-13-dtls“

es kann nur ein handshake zur zeit stattfinden
psk wird nur benötigt bis pre master secret berechnet
am ende des handshakes wird alter psk vernichtet und neuer generiert
kann durch uri /psk über sichere leitung abgerufen werden so dass weitere handshakes durchgeführt werden können

für hello verify cookie: cmac(client-ip + clienthello)

verschlüsselung von finish mit seq_num 0. erste app daten beginnen also bei seq_num 1.
seq_num wird im ram gehalten da bei flash update bei jedem paket hoher verschleiß.
sollte seq_num je 0 sein, wird sie mit der seq_num der anfrage „wiederhergestellt“ (erläuterung: ist nicht exakt)

6.1.5 Update-Funktion

flasher beschreiben: überschreibt den programmcode jedoch nicht die vom blaster generierten gerätedaten

6.2 Client

6.3 Testumgebung

flashen zunächst generell an ttyUSB1

border-router per „make border“ an ttyUSB1

sniffer per „make listen2“ an ttyUSB3
„make listen“ würde direkt in wireshark pipen. funktioniert nicht

server nach initialem flash -> update per client über coap

Kapitel 7

Vergleich

7.1 Headercompression

lala soso

7.2 Ciphersuit

CBC-MAC anstatt HMAC mit SHA256

lala soso

Kapitel 8

Fazit

generell machbar

vor und nachteile abwägen

trotz dtls-cookie -> zustand durch block 1
ausblick -> cookie unterteilen
ersten 2 byte als 4 stelligen hex wert als query anhängen -> hash von den ersten 32 byte

contiki speicher weiter optimieren - stacks notwendig ?

Akronyme

6LoWPAN IPv6 over Low power Wireless Personal Area Network, S. 2, 13

ACK Acknowledgement, S. 13
AEAD Authenticated Encryption with Associated Data, S. 5, 19
AES Advanced Encryption Standard, S. 5, 17–20, 22

CBC Cipher Block Chain, S. 5
CCM Counter with CBC-MAC, S. 5, 17–19
CMAC Cipher-based Message Authentication Code, S. 20
CoAP Constrained Application Protocol, S. 1, 2, 5, 11–16, 22, 23
CTR Counter, S. 5

DoS Denial of Service, S. 9
DTLS Datagram Transport Layer Security, S. 1–3, 5–7, 10–15, 17, 19, 21, 23, 24

ECC Elliptic Curve Cryptography, S. 18, 19

HMAC Keyed-Hashing for Message Authentication, S. 18, 20

IETF Internet Engineering Task Force, S. 1

MAC Message Authentication Code, S. 5, 12, 13, 19

NIST National Institute of Standards and Technology, S. 19

PIN Persönliche Identifikationsnummer, S. 21
PRF Pseudo-Random-Funktion, S. 18–21
PSK Pre-Shared Key, S. 5, 8, 17, 18, 20–22

RAM Random-Access Memory, S. 3
ROM Read-Only Memory, S. 3

SHA Secure Hash Algorithm, S. 17, 18, 20

TCP Transmission Control Protocol, S. 1, 7

TLS Transport Layer Security, S. 1–3, 7, 10, 12, 14–16

UDP User Datagram Protocol, S. 1, 7, 12, 13, 15

URI Uniform Resource Identifier, S. 13, 14

UUID Universally Unique Identifier, S. 21

WoT Web of Things, S. 1

Glossar

Ciphersuit
: Gruppe aus 4 Algorithmen, die für Schlüsselaustausch, Authentifizierung, Hash und Verschlüsselung verwendet werden
S. 2, 3, 5, 8, 12, 21

Fragment
: Beschreibt einen Teil eines Ganzen
S. 2

Fragmentierung
: Beschreibt die Aufteilung eines Ganzen in kleinere Teile
S. 2

Handshake
: Beschreibt die Aushandlung von Sicherheitsparametern um eine sichere Verbindung herzustellen
S. 2, 3, 7

Man-in-the-middle-Angriff
: Angriff auf eine Kommunikationsverbindung zwischen zwei Parteien, bei dem ein Angreifer die vollständige Kontrolle über den Datenverkehr der Verbindung übernimmt
S. 18, 21

MC13224v
: ARM7TDMI-S Microcontroller der Firma Freescale Semiconductor, Inc
S. 3, 5, 22, 25, 26

RSA-Verfahren
: Asymmetrisches kryptographisches Verfahren zur Verschlüsselung und Signatur, das nach seinen Erfindern Rivest, Shamir und Adleman benannt ist
S. 17

Literaturverzeichnis

[Bla+06] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk und B. Moeller. *Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)*. RFC 4492 (Proposed Standard). Internet Engineering Task Force, Mai 2006. URL: *http://tools.ietf.org/html/rfc4492*.

[BS13] C. Bormann und Z. Shelby. *Blockwise transfers in CoAP*. Internet-Draft. Internet Engineering Task Force, Juni 2013. URL: *http://tools.ietf.org/html/draft-ietf-core-block*.

[Coco9] Chris K Cockrum. *Implementation of an Elliptic Curve Cryptosystem on an 8-bit Microcontroller*. Apr. 2009. URL: *http://cockrum.net/Implementation_of_ECC_on_an_8-bit_microcontroller.pdf*.

[Con13] Contiki-Community. *Contiki - The Open Source OS for the Internet of Things*. Mai 2013. URL: *http://www.contiki-os.org/*.

[DR08] T. Dierks und E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246 (Proposed Standard). Internet Engineering Task Force, Aug. 2008. URL: *http://tools.ietf.org/html/rfc5246*.

[ET05] P. Eronen und H. Tschofenig. *Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)*. RFC 4279 (Proposed Standard). Internet Engineering Task Force, 2005. URL: *http://tools.ietf.org/html/rfc4279*.

[Fre13] Freescale Semiconductor. *MC1322x - Advanced ZigBee™ - Compliant Platform-in-Package (PiP) for the 2.4 GHz IEEE® 802.15.4 Standard*. Mai 2013. URL: *http://www.freescale.com/files/rf_if/doc/data_sheet/MC1322x.pdf*.

[HB12] K. Hartke und O. Bergmann. *Datagram Transport Layer Security in Constrained Environments*. Internet-Draft. Internet Engineering Task Force, Juli 2012. URL: *http://tools.ietf.org/html/draft-hartke-core-codtls-02*.

[Hee13] Dimitri van Heesch. *Doxygen*. Aug. 2013. URL: *http://www.stack.nl/~dimitri/doxygen*.

[Int13a] Internet Assigned Numbers Authority (IANA). *Service Name and Transport Protocol Port Number Registry*. Aug. 2013. URL: *http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml*.

[Int13b] Internet Assigned Numbers Authority (IANA). *Transport Layer Security (TLS) Parameters*. Juli 2013. URL: *http://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml*.

[KK13] Harald Kipp und Sven Köhler. *ARM GCC Inline Assembler Cookbook*. Aug. 2013. URL: *http://www.ethernut.de/en/documents/arm-inline-asm.html*.

[Kov13] Matthias Kovatsch. *Erbium (Er) REST Engine and CoAP Implementation for Contiki*. ETH Zurich. Mai 2013. URL: *http://people.inf.ethz.ch/mkovatsc/erbium.php*.

[LAN11] LAN/MAN Standards Committee and IEEE Computer Society. *Low-Rate Wireless Personal Area Networks (LR-WPANs)*. IEEE 802.15.4-2011. IEEE Standards Assocoation, Juni 2011. URL: *http://standards.ieee.org/getieee802/download/802.15.4-2011.pdf*.

[MB12] D. McGrew und D. Bailey. *AES-CCM Cipher Suites for Transport Layer Security (TLS)*. RFC 6655 (Proposed Standard). Internet Engineering Task Force, Juli 2012. URL: *http://tools.ietf.org/html/rfc6655*.

[McG08] D. McGrew. *An Interface and Algorithms for Authenticated Encryption*. RFC 5116 (Proposed Standard). Internet Engineering Task Force, Jan. 2008. URL: *http://tools.ietf.org/html/rfc5116*.

[McG+11] D. McGrew, D. Bailey, M. Campagna und R. Dugal. *AES-CCM ECC Cipher Suites for TLS*. Internet-Draft. Internet Engineering Task Force, Okt. 2011. URL: *http://tools.ietf.org/html/draft-mcgrew-tls-aes-ccm-ecc-06*.

[Nat04] National Institute of Standards and Technology (NIST). *Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality*. NSP 800-38C. National Institute of Standards und Technology, Mai 2004. URL: *http://csrc.nist.gov/publications/nistpubs/800-38C/SP800-38C.pdf*.

[Red13] Redwire, LLC. *Econotag: mc13224v development board with on-board debugging*. Mai 2013. URL: *http://www.redwirellc.com/store/node/1*.

[RM12] E. Rescorla und N. Modadugu. *Datagram Transport Layer Security Version 1.2*. RFC 6347 (Proposed Standard). Internet Engineering Task Force, Jan. 2012. URL: *http://tools.ietf.org/html/rfc6347*.

[She+12] Z. Shelby, K. Hartke, C. Bormann und B. Frank. *Constrained Application Protocol (CoAP)*. Internet-Draft. Internet Engineering Task Force, 2012. URL: *http://tools.ietf.org/html/draft-ietf-core-coap-13*.

[Son+06] JH. Song, R. Poovendran, J. Lee und T. Iwata. *The AES-CMAC Algorithm*. RFC 4493 (Proposed Standard). Internet Engineering Task Force, Juni 2006. URL: *http://tools.ietf.org/html/rfc4493*.

[TKK13] H. Tschofenig, S.S. Kumar und S. Keoh. *A Hitchhiker's Guide to the (Datagram) Transport Layer Security Protocol for Smart Objects and Constrained Node Networks*. Internet-Draft. Internet Engineering Task Force, Juli 2013. URL: *http://tools.ietf.org/html/draft-tschofenig-lwig-tls-minimal-03*.

[WHF03] D. Whiting, R. Housley und N. Ferguson. *Counter with CBC-MAC (CCM)*. RFC 3610 (Proposed Standard). Internet Engineering Task Force, Sep. 2003. URL: *http://tools.ietf.org/html/rfc3610*.

Abbildungsverzeichnis