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, Oktober 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 1990 Jahre das Secure Sockets Layer Protocol (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 SSL weiterentwickelt, und schließlich in Transport Layer Security Protocol (TLS) [DR08] umbenannt wurde, nahm die Beliebtheit von UDP, unter anderem im Bereich der Onlinespiele, zu [RM12, Kapitel 1]. Um dort ebenfalls eine sichere Datenübertragung zu ermöglichen, 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 Protocol (DTLS) [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 Endgeräte haben nur sehr wenig Ressourcen zur Verfügung. Dies betrifft neben wenig Rechenleistung und Speicher, der vielfach unter 100 KiB liegt, auch den Energievorrat, der über eine Batterie realisiert wird. Etabliert hat sich dort das Constrained Application Protocol (CoAP) [She+12] über UDP aufgrund seines schlanken Designs. Passend zu UDP ist DTLS, das sich, aufgrund seines Umfangs, jedoch nicht so einfach auf den vorher beschriebenen 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

. .

warum dieses kapitel und was wird hier gemacht

1.2.1 Datagram Transport Layer Security in Constrained Environments

Im IETF-Draft „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.

Als eines der Hauptprobleme nennen die Autoren die geringe Paketgröße in Netzen, die den Funkstandard IEEE 802.15.4 [LAN11] verwenden, da hier die Nutzdaten auf eine Länge von 127 Byte pro Paket beschränkt sind. Insbesondere beim Aufbau der sicheren Verbindung (Handshake) müssen viele Daten ausgetauscht werden, was bei der Verwendung von DTLS die Paketgröße überschreiten würde. Durch Internet Protocol, Version 6 (IPv6) [DH09], das in eingeschränkten Umgebungen mit Hilfe von IPv6 over Low power Wireless Personal Area Network (6LoWPAN) [Mon+07] realisiert wird, würde sich das Problem durch eine Nutzung der IP-Fragmentierung lösen lassen. Das führt aber bei Verlust einzelner Pakete, zu einem neuen Versand aller IP-Fragmente, und erzeugt somit umfangreichen Datenverkehr, 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. Dafür schlagen die Autoren eine Stateless Header Compression vor. Bei Nutzung von CoAP wäre es auch möglich, den Verbindungsaufbau über CoAP zu realisieren. Dadurch ist die Zuverlässigkeit des Transports 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 beispielsweise im Elliptic Curve Diffie-Hellman 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. Auch kann der Verbindungsaufbau schon durchgeführt werden, bevor Anwendungsdaten zum Übertragen vorhanden sind. Liegen dann Anwendungsdaten vor, können diese ohne Verzögerung übertragen werden.

Um Speicher zu sparen, müssen auch die Anzahl der sicheren Verbindungen begrenzt werden und 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 IETF-Draft „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 bei einem Handshake von der Wahl des Cipher-Suites 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 Clients erhalten kann. Je klarer die Position und die Umgebung des Sensors sind, desto weniger flexibel muss dieser sein, woraus eine spezialisierte Implementierung resultiert, die den Aufwand und die Codegröße reduziert.

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

Kernstück der Arbeit 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 Prototyps zeigen sie dort auf, welche grundlegenden Teile von DTLS, ohne Berücksichtigung der Cipher-Suite 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 beispielsweise Hash-Funktionen und anderen für TLS notwendigen Berechnungen ausgewertet, wie sie in unterschiedlichen Cipher-Suites verwendet werden.

Abschließend stellen sie fest, dass sich TLS/DTLS durchaus auf eingeschränkte Umgebungen zuschneiden lässt, wobei mehr Flexibilität aber zu größerem Programmcode führt. hier vielleicht ein paar Kennzahlen zitieren?

1.3 Ziel dieser Arbeit

Ziel dieser Arbeit soll es sein, DTLS so weit anzupassen, dass es sich in eingeschränkten Umgebungen, insbesondere auf einem Redbee Econotag [Red13] mit dem MC13224v [Fre13] Mikrocontroller, 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 Cipher-Suites angewendet werden können.

1.4 Vorgehensweise

Im Vordergrund soll die Implementierung eines Sicherheitsprotokolls stehen, das sich an den Prinzipien von DTLS orientiert und Vorschläge aus dem IETF-Draft „Datagram Transport Layer Security in Constrained Environments“ [HB12] aufgreift. Ein besonderes Interesse liegt darauf, den Handshake über CoAP [She+12] zu realisieren und somit einige in DTLS eingefügte Mechanismen überflüssig zu machen. Die Implementierung wird im Anschluss durch einen Vergleich mit DTLS evaluiert, wobei folgende Punkte eine

Rolle spielen: Volumen des generierten Datenaufkommens, Umfang des Quellcodes und die vom Protokoll benötigte Speichermenge.

Die Implementierung besteht aus dem Client auf einem gängigen PC, 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 [Fre13] Mikrocontroller 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 eine Cipher-Suite realisiert werden. Für diese dient „TLS_PSK_DHE_WITH_AES_128_CCM_8“ aus RFC 6655 [MB12] als Grundlage, wobei dort, aufgrund von Erfahrungen aus dem Bachelorprojekt, Anpassungen notwendig sind. Durch diese wird für den Verbindungsaufbau ein Schlüsselaustausch vorgegeben, wobei zusätzlich ein Pre-Shared Key (PSK) verwendet wird, damit sich die Kommunikationspartner gegenseitig authentifzieren können. Ein Verbindungsaufbau ist somit nur möglich, wenn beide Kommunikationspartner vorher einen gemeinsamen PSK vereinbart haben. Die Verschlüsselung der Anwendungsdaten erfolgt dann im Modus „Authenticated Encryption with Associated Data (AEAD)“ [McG08] wobei sich hier „Counter with CBC-MAC (CCM)“ [WHF03] aufgrund der Hardwarevorraussetzungen am besten eignet. Dieser besteht aus einer Verschlüsselung der Daten mit 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.

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 Übertragungen betrachtet, sondern auch Paketverluste mit einbezogen. Verglichen wird auch der notwendige Speicherbedarf bei Verbindungsaufbau und Übertragung der Anwendungsdaten. Ebenso soll der Umfang des zugrunde liegenden Quellcodes bewertet werden.

1.5 Struktur

Struktur beschreiben

... ...

Kapitel 2

DTLS

motivation für das kapitel

Das Sicherheitsprotokoll TLS [DR08] wird im Allgemeinen mit dem stromorientierten TCP verwendet. Wurde eine Verbindung mit TCP hergestellt, können Daten beliebieger Größe in jede Richtung übertragen werden. TCP sorgt dafür, 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 2.1 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 vier 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, gegenüber TLS, 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 fehlerfreien 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;
} DTLS_Record;
```

Abbildung 2.1 Header des Record-Layer-Protokolls von DTLS

2.1 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 2.2 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 zu gewährleisten, dass diese vollständig, und in der richtigen Reihenfolge, bearbeitet werden. Da auf eine Fragmentierung der UDP-Pakete auf IP-Ebene vermieden werden soll, und somit die Paketgröße begrenzt ist, 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, wobei 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
} Handshake;
```

Abbildung 2.2 Header des Handshake-Protokolls von DTLS

Der für einen Handshake durchzuführende Nachrichtenaustausch ist in vollständiger Form in Abbildung 2.3 aufgeführt. Die mit * markierten Pakete werden hier kurz erklärt, spielen aber im weiteren Verlauf keine Rolle, da die Authentifizierung durch den PSK realisiert werden soll und auf die Zertifikate verzichtet wird, um Ressourcen zu sparen.

Eingeleitet wird der Handshake mit einer Nachricht vom Typ ClientHello, in der der Client seine Möglichkeiten bekannt gibt. Dazu gehören u. a. die unterstützten Protokollversionen, Cipher-Suites und Kompressionsmethoden. Während der Server bei TLS nun direkt mit einer ServerHello-Nachricht und weiteren Handshake-Paketen antworten kann, lässt sich das bei DTLS so nicht realisieren. Da UDP kein verbindungsorientiertes Protokoll ist, können Pakete mit gefälschtem Absender versendet werden. Auf diese Art könnte

```
                    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 2.3 Nachrichtenaustausch während eines DTLS-Handshakes

Angriff vom Typ Denial-of-Service (DoS) 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 validieren, wurde in DTLS ein Cookie ergänzt. Dieser wird aus dem ClientHello generiert und als Antwort an den Client 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 validiert. Trotz dieses Verfahrens ist es jedoch möglich, den Energievorrat des Servers durch unzählige Anfragen zu redzuieren, da auch die erste Anfrage (ohne Cookie) bearbeitet werden muss. Lediglich der dafür notwendige Aufwand ist geringer. Auch schützt dieses Verfahren nicht gegen einen Man-in-the-middle-Angriff.

Im ServerHello gibt der Server bekannt, welche der vom Client genannten Möglichkeiten, wie beispielsweise die unterstützten Cipher-Suites, ausgewählt wurden. Folgen können dann ein Zertifikat, Daten für einen Schlüsselaustausch sowie eine Anfrage für das Zertifikat des Clients. Abschließend folgt ein ServerHelloDone, um dem Client 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 er-

möglichen, das Zertifikat des Clients 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 ausgetauscht, die wieder zum Handshake-Protokoll gehört. Diese enthält einen Hashwert von allen bisher ausgetauschten Nachrichten und wird mit den Sicherheitsparametern der neuen Epoche verschlüsselt. So wird der Handshake verifiziert und die neuen Sicherheitsparemeter auf Korrektheit geprüft.

anschließend können Nutzdaten über das App-Data-Protokoll versendet werden

es werden die gerade ausgehandelten Sicherheitsparameter genutzt

2.2 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 2.4) enthält neben dem AlertLevel, welches *warning* (1) oder *fatal* (2) sein kann, die Beschreibung des Fehlers. Während Fehler der Stufe *fatal* zu einem unmittelbaren Verbindungsabbruch führen, sind Fehler der Stufe *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.

wie auch alle anderen pakete -> sicherheitsparameter der aktuellen epoche, verschlüsselt

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

Abbildung 2.4 Header des Alert-Protokolls von DTLS

Kapitel 3

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.

3.1 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 3.1 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 3.1 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 3.2). 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 Cipher-Suites 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.

3.2 Handshake

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

Wie in Kapitel 2 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 kommen, 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 3.2 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 3.2 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, um sowohl Daten als auch Energie zu sparen, da er im Klartext in die CoAP-Anfrage eingefügt wird. DTLS-Sessions, die während eines Handshakes erzeugt werden, bilden Sub-Ressourcen. Gemäß CoAP [She+12, Abschnitt 5.8.2], soll eine POST-Anfrage, die eine Ressource erzeugt, mit einem 2.01 Created und der Location-Path-Option beantwortet werden, die den neuen URI enthält. Auf die Location-Path-Option wird hier verzichtet, da diese nicht zwingend ist, und die Session-ID im ServerHello-Paket enthalten ist, womit sich der neue URI berechnen lässt. Die Verwendung der Location-Path-Option hätte den Nachteil, dass diese bei einer blockweisen Übertragung, in jedem Block wiederholt werden würde. Dieser Nachteil tritt bei einer CoAP-Anfrage ebenfalls ein, wird hier jedoch akzeptiert, da es so bei einer blockweisen Übertragung frühzeitig möglich ist, die Gültigkeit der Ressource zu überprüfen. Während die Session-ID somit im ClientHello-Paket überflüssig ist und entfernt wird, bleibt diese im ServerHello-Paket enthalten. Da-

durch ist auch fest definiert, dass ein ClientHello-Paket an die Haupt-Ressource die Erstellung einer neuen Sub-Ressource bewirkt, während ein ClientHello-Paket an eine Sub-Ressource die Wiederaufnahme einer Session bewirkt.

Während in einem gewöhnlichen DTLS-Handshake jedes einzelne DTLS-Handshake-Paket sowohl den DTLS-Record-Header als auch den DTLS-Handshake-Header enthält, ist dies hier nicht mehr notwendig, da mehrere DTLS-Handshake-Pakete innerhalb eines CoAP-Pakets enthalten sind. Der DTLS-Record-Header ist einmalig vor jedem CoAP- oder Alert-Paket enthalten. Um die DTLS-Handshake-Pakete innerhalb eines CoAP-Pakets voneinander abzugrenzen, dient nun der DTLSConten-Header (Abbildung reffig:com_content_header).

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

Abbildung 3.3 Komprimierter 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 Header folgenden Bytes der Länge, wobei der Wert 0 die Länge 0 direkt definiert. Neben den in 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 „4.00 Bad Request“. Ein Beispiel dafür ist in Abbildung 3.4 zu sehen.

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 Pa-

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

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

                           <-------- 4.00 Bad Request
                                        Alert(handshake_failure)
```

Abbildung 3.4 Nachrichtenaustausch während eines TLS / DTLS Handshakes über CoAP

ketversands 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 3.2 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 3.2 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 CipherSuite wird dadurch die in TLS definierte Länge des Finished-Pakets von 12 vergrößert, da unter Umständen Zusatzinformationen wie Nonce und MAC hinzukommen.

Kapitel 4

Definition des Cipher-Suites

Neben den grundlegenden Anpassungen im DTLS-Record-Layer und der Änderung des Handshakes ist es notwendig eine geeignete Cipher-Suite 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. Die Cipher-Suite 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 Cipher-Suites 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 Cipher-Suites definiert sind, die zur Berechnung des Schlüssels das rechenaufwendige RSA-Verfahren benutzen, sind in Kapitel 4 einige Cipher-Suites 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 3.2). 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 Cipher-Suite 1 und 2 nicht weiter betrachtet werden.

Im Unterschied zu Cipher-Suite 3 wird bei Cipher-Suite 4 nicht nur der PSK verwendet um den Schlüs-

sel 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 Cipher-Suite 3 kommt somit nicht in Frage, womit Cipher-Suite 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 eine neue Cipher-Suite definiert.

4.1 TLS_PSK_ECDH_WITH_AES_128_CCM_8

In Ahnlehnung an das Cipher-Suite 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 die Cipher-Suite nun ihren Namen:

- TLS_PSK_ECDH_WITH_AES_128_CCM_8

Da dieses keine offizielle Cipher-Suite 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 4.1). Während die Epoche und die Sequenznummer

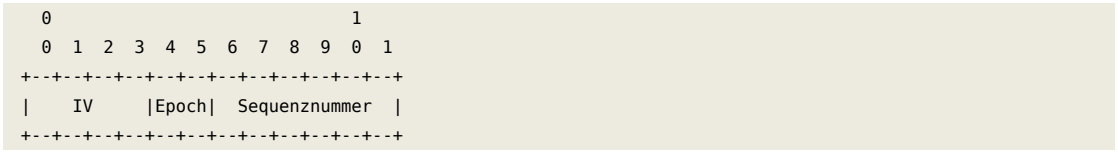

Abbildung 4.1 Nonce für AES-CCM

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 diese Cipher-Suite 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 4.2 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 4.2 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 3.1 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 5

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].

Die im vorigen Kapitel definierte Cipher-Suite 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 können die Daten, mit Hilfe des 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 ursprünglichen PSK wieder aktiviert.

5.1 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, 83,73 KiB. Diese Daten teilen sich gemäß Abbildung 5.1 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. Außerdem wurde „REST_MAX_CHUNK_SIZE“ in der Contiki-App „Erbium“ von 128 auf 48 Byte reduziert, wodurch das Datensegment weniger Speicher benötigt. Diese Konstante definiert die maximale Größe der Anwendungsdaten, die in einem Contiki-Paket untergebracht werden können, und stellt somit sicher, dass jedes CoAP-Paket in ein einzelnes IP-Paket passt.

Beschreibung	**Standard**	**Angepasst**
Programm	58760 Byte	58760 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	21064 Byte	20744 Byte
Heap	4096 Byte	16 Byte
Gesamt	85744 Byte	82368 Byte
	83,73 KiB	80,44 KiB

Abbildung 5.1 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 83,73 KiB auf 80,44 KiB reduziert (siehe Abbildung 5.1). Somit stehen für die Umsetzung von DTLS ~15,5 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 kann von einem Clienten benutzt werden, um größere Datenmengen, in einer CoAP-Anfrage, in Blöcke zu unterteilen, damit es nicht zu einer Fragmentierung auf IP-Ebene kommt. Laut Aussage von Matthias Kovatsch wird „auf die atomare Variante aus Platzgründen verzichtet, da man Block1 ganz einfach im Resource-Handler lösen kann„. Da diese Option für den DTLS-Handshake generell benötigt wird, und ohne Code-Duplizierung auch anderen Ressourcen zur Verfügung stehen soll, wird sie in ähnlicher Form wie die Separate-Option implementiert. Das Separate-Modul bietet Methoden an, um den Client, während der Bearbeitung einer Anfrage, zu informieren, dass die Bearbeitung einige Zeit dauert, und die Beantwortung der Anfrage später fortzusetzen. In diesem Sinne bietet das Block-1-Modul eine Methode an, mit der die Parameter der Block-1-Option überprüft und bearbeitet werden, wobei bei Bedarf die entsprechenden Fehler generiert werden, oder die erhaltenen Daten auf Wunsch zusammengesetzt werden. Genau wie das Separate-Modul kann das Block1-Modul optional in einer Ressource benutzt werden. Dabei ist nach wie vor ein Empfang von Daten ohne Block-1-Option möglich. Durch den Rückgabewert der Methode, lässt sich in der Ressource entscheiden, ob schon die einzelnen Datenblöcke bearbeitet werden, oder erst die vollständige Nachricht, nach Erhalt aller Blöcke. Damit das Separate-Modul auch in Kombination mit dem Block-1-Modul genutzt werden kann, wurde das Separate-Modul entsprechend angepasst um die Block-1-Option zu berücksichtigen.

Während bisher allgemeine Anpassungen von SmartAppContiki bzw. dem darin enthaltenen CoAP 13 beschrieben wurden, folgt in den nächsten vier Abschnitten die Erläuterung von vier implementierten Contiki-Apps, welche für die Realisierung von DTLS benutzt werden. Die Implementierung von DTLS wird dann im 5. Abschnitt erläutert, wonach abschließend noch eine Update-Funktion erläutert wird, die für DTLS nicht notwendig ist, aber dessen Umfeld berücksichtigt.

5.1.1 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 5.2 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 5.2 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 5.2 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 Pro-

gramm „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.

5.1.2 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.

5.1.3 Contiki-App: „aes“

Um die AES-Funktionen des MC13224v [Fre13] zu nutzen, dient diese Contiki-App. Die bereits beschriebene Cipher-Suite verwendet in der PRF AES-CMAC [Son+06]. Außerdem wird für die Verschlüsselung AES-CCM [WHF03] verwendet. Beide Verfahren werden vom MC13224v nicht direkt unterstützt. Bereitgestellt wird nur der reine AES-Verschlüsselungsprozess im CTR- und CBC-Mode.

Damit die AES-Hardware genutzt werden kann, muss diese zunächst initialisiert werden. Die dafür notwendige Methode wurde zum Großteil aus dem Bachelorprojekt GOBI übernommen und leicht modifiziert.

Neben der Aktivierung der AES-Hardware wir dort ein Selbsttest mit einem internen Schlüssel durchgeführt. Ist dieser erfolgreich, werden die beiden Modi CTR und CBC, für die spätere Nutzung, aktiviert.

Die AES-Berechnungen selbst werden durch Übertragen der notwendigen Daten in Register des Mikrocontrollers durchgeführt. Diese sind auf Speicheradressen abgebildet, die wiederum in Konstanten in der Bibliothek des Mikrocontrollers hinterlegt sind. Da es sich um eine 128-bit-Verschlüsselung handelt, der Mikrocontroller aber nur in 32 Bit arbeitet, sind für jeden Wert vier Register notwendig. Diese sind jeweils von 0 bis 3 durchnummeriert, was in der folgenden Beschreibung durch <X> dargestellt wird. Zusätzlich sind zwei Register notwendig, um den Verschlüsselungsprozess zu starten und auswerten zu können.

KEY<X>
: Schlüssel zur Ver- und Entschlüsselung. Verbleibt so lange im Register, bis das AES-Modul zurückgesetzt, oder das Register überschrieben wird.

DATA<X>
: Datenpaket mit Klar- oder Geheimtext.

CTR<X>
: Zähler für den CTR-Mode. Dieser wird nicht automatisch erhöht und muss somit vor jeder Berechnung gesetzt werden.

CTR<X>_RESULT
: Ergebnis der CTR-Berechnung. Dafür wurde der hinterlegte Zähler verschlüsselt und durch die XOR-Funktion mit dem Datenpaket verknüpft. Dieses Register kann nur gelesen werden.

CBC<X>_RESULT
: Ergebnis der CBC-Berechnung. Dafür wurde das Datenpaket durch die XOR-Funktion mit dem Ergebnis der letzten Verschlüsselung verknüpft und dann verschlüsselt. Dieses Register kann nur gelesen werden.

MAC<X>
: Kann mit einem Initialisierungsvector belegt werden, der für die CBC-Berechnung herangezogen wird.

CONTROL0bits
: Enthät u. a. ein Bit, durch das die Verwendung des Initilalisierungsvectors gekennzeichnet wird. Außerdem ist ein Bit dafür vorgesehen, den Ver- und Entschlüsselungsprozess zu starten.

STATUSbits
: Enthät u. a. ein Bit, das kennzeichnet, ob die aktuelle Berechnung abgeschlossen ist.

Da der Ent- und Verschlüsselungsprozess derselbe ist, reicht eine Methode für die Umsetzung von AES-CCM aus. Diese arbeitet in-place, was bedeutet, dass die Daten direkt an ihrer Position im Speicher konvertiert werden, und zusätzlich nur eine konstante, von der Datenmenge unabhängige, Menge Speicher benötigt wird. Während für die Verschlüsselung ein Aufruf der Methode ausreicht, da die Daten verschlüsselt werden, und der MAC berechnet wird, sind für die Entschlüsselung zwei Aufrufe notwendig. Zunächst werden die Daten entschlüsselt, wobei automatisch ein neuer MAC, auf Basis des Geheimtextes, generiert wird. Dieser hat jedoch keinen Nutzen. Im 2. Schritt wird die Funktion erneut aufgerufen, um ausschließlich den MAC zu generieren, damit dieser mit dem erhaltenen verglichen werden kann. Das führt im 1. Schritt zwar zu

unnötigen Berechnungen, verlangsamt aber den Prozess nicht, da Entschlüsselung und MAC-Berechnung parallel in der Hardware durchgeführt werden. Der Vorteil liegt hier in der geringen Programmgröße.

Die Methode zur Berechnung der CMAC ist so gestaltet, dass die Berechnung in mehreren Schritten erfolgen kann, solange die Länge der übergebenen Daten ein Vielfaches von 16 Byte (128 Bit) beträgt. Erst durch Setzen des letzten Parameters, welcher den Abschluss signalisiert, ist die Datenlänge beliebig, und die Berechnung wird gemäß CMAC-Vorgabe abgeschlossen.

5.1.4 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 [Coc09], 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 24 und 32 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 ~32 Byte ab und die Berechnungsgeschwindigkeit nimmt wesentlich zu. Anders verhält es sich bei der Addition, da Werte unterschiedlicher Größe addiert werden. Notwendig sind hier 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 88 Byte.

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.

5.1.5 Contiki-App: „er-13-dtls“

Um diese Contiki-App zu realisieren, sind zunächst einige Anpassungen in er-coap-13 notwendig. Um nach wie vor einen Betrieb ohne DTLS realsisieren zu können, werden diese Anpassungen nur dann aktiv, wenn die Compiler-Anweisung WITH_DTLS gesetzt ist, was im Makefile für ein Contiki-Programm durch „CFLAGS += -DWITH_DTLS=1“ realisiert werden kann. Bei der Verwendung von DTLS wird der Port gemäß [Int13a, Seite 93] von 5683 auf 5684 geändert. Außerdem wird der Datenverkehr über DTLS-Funktionen geleitet, die den Record-Layer realisieren und die Daten bei Bedarf ent- oder verschlüsseln. Kernstück ist die Ressource „/dtls“, die für die Durchführung des Handshakes in CoAP eingebunden wird.

Für die Realisierung des Recory-Layers und der DTLS-Ressource sind einige Module notwendig, die im Folgenden zunächst beschrieben werden.

er-dtls-13-alert.[**h|c**]

Stellt zwei Funktionen für den Versand einer Alert-Nachricht zur Verfügung. Während die eine, ohne Verwendung con CoAP, eine Nachricht an den Kommunikationspartner sendet, konfiguriert die andere eine CoAP-Antwort. Dieser Unterschied ist notwendig, da einige Fehler während des Handshakes auftauchen, und per CoAP kommentiert werden, damit es nicht zu einer neuen Anfrage kommt. Würden diese Nachrichten direkt mit dem Record-Protokoll versand, müsste sich der Empfänger darum kümmern, die Anfrage aus dem CoAP-Layer zu entfernen.

er-dtls-13-data.[**h|c**]

Erstellt und verwaltet sessionspezifische Daten. Die Anzahl der Sessions ist hier auf maximal zehn begrenzt. Damit die dafür notwendigen 1400 Byte nicht dauerhaft den RAM-Speicher belegen, sind diese im Flash-Speicher abgelegt. Die Ausnahme bilden hier die beiden Werte für die Sequenznummern zum Lesen und Schreiben. Während letztere bei jedem Datenaustausch geändert werden, werden die grundlegenden Session-Daten nur während des Handshakes geschrieben, so dass hier die Nutzung des Flash-Speichers sinnvoll ist. Dafür wird der RW-Block verwendet, der durch die Contiki-App flashstore zur Verfügung gestellt wird. Die Sessions sind dort in einem Array hinterlegt, wobei die genutzten Stellen entsprechend gekennzeichnet sind. Dadurch werden bei der Suche nach Sessions unter Umständen auch leere Stellen durchlaufen, jedoch bleibt der Aufwand erspart die Liste zu defragmentieren oder Zeiger einer verketteten Liste zu aktualisieren. Da pro Session zwei Keyblöcke benötigt werden, falls es zu einem erneuten Handshake kommt, sind diese in einem separaten Array mit der Länge 20 hinterlegt. Der Index der Keyblöcke ergibt sich dabei aus dem Index der Session. Liegt diese im ersten Array an Index i, Sind die dazu gehörenden Keyblöcke an Index $i \cdot 2$ und $i \cdot 2 + 1$ hinterlegt. Dabei liegt der derzeit gültige Keyblock immer an Index $i \cdot 2$. Bei einer Weiterentwicklung der Epoche, wird der 2. Keyblock an den Index des ersten kopiert. Da die Sequenznummern im RAM-Speicher abgelegt sind, gehen diese bei einem Batteriewechsel, oder Neustart des Endgeräts, verloren, womit die Session nicht fortgesetzt werden kann. Da es keine sinnvolle Lösung gibt, diese ohne Sicherheitslücken wiederherzustellen, wird bei einem Start des Endgeräts der RW-Block des Flash-Speichers zurückgesetzt, wodurch alle Sessiondaten gelöscht werden. Bei einer folgenden Anfrage wird der Client zunächst eine Alert-Nachricht bekommen, was aber dann zu einem neuen Handshake führt.

er-dtls-13-prf.[**h|c**]

Enthält die im Cipher-Suite definierte PRF. Die größte Datenmenge wird für die Berechnung des Master-Secrets benötigt. Dort gehen 153 Byte in die Berechnung ein. Dieses ist ausreichend klein um die Berechnung mit einem Funktionsaufruf durchzuführen, womit der Programmcode klein gehalten wird, da kein Zustand für eine Fortsetzung der Berechnung gespeichert und genutzt werden muss.

er-dtls-13-psk.[**h|c**]

Verwaltet den PSK und generiert bei Bedarf einen neuen. Der PSK wird im Vorfeld durch das Programm Blaster generiert und im Flash-Speicher abgelegt. Soll ein neuer PSK generiert werden, um den Werks-PSK zu deaktivieren, wird ein Byte im RW-Block des Flash-Speichers gesetzt und dort ebenfalls ein neuer PSK hinterelgt. Wird der PSK abgerufen, kann anhand des gesetzten Bytes erkannt werden, ob der Werks-PSK gilt oder ein neuer Verfügbar ist. Kommt es zu einem Batteriewechsel oder Neustart des

Endgeräts, ist der Werks-PSK wieder gültigt, da der RW-Block zurückgesetzt wurde.

er-dtls-13-random.[**h**|**c**]

Bietet Funktionen für die Generierung von Zufallszahlen an. Dieses Modul benutzt das MACA_RANDOM Register des MC13224v, das durch Contiki bereits initialisiert wurde. Durch Auslesen des Registers können beliebig viele Zufallswerte erzeugt werden.

Im Record-Layer wird das Modul für die Sessionverwaltung und das Modul für die Alert-Nachrichten genutzt. Bei der Bearbeitung einer Nachricht, werden die benötigten Session-Daten abgerufen und zur Bearbeitung der Nachricht genutzt. Sollten dabei Fehler auftreten, wird eine Alert-Nachricht an den Client gesendet und das Paket verworfen. Zu beachten ist bei eingehenden Daten, dass sowohl Handshake-Nachrichten als auch Anwendungsdaten CoAP-Pakete enthalten. Während Handshake-Nachrichten in Epoche 0, ohne Verschlüsselung, gestattet sind, ist dies bei Anwendungsdaten nicht erlaubt. Bei Handshake-Nachrichten wird somit sichergestellt, dass die erste, im CoAP-Paket enthaltene, URI „dtls“ entspricht, da ein Angreifer ansonsten beliebige CoAP-Anfragen als Handshake-Nachricht tarnen könnte.

In der CoAP-Ressource „/dtls“ werden schließlich Handshake-Nachrichten bearbeitet. Falls es hier zu Fehlern kommt wird der Client darüber in einer CoAP-Antwort, die durch das Modul für die Alert-Nachrichten generiert wird, informiert. Ein Fehler resultiert in einem Abbruch des Ressource-Handlers, was durch einen Sprung an das Ende der Methode realisiert wird, damit globale Variablen zurückgesetzt werden können.

Um Speicher zu sparen, und die Block-1-Option von CoAP einfach nutzen zu können, wird nur ein Handshake zur Zeit gestattet. Dafür wird bei einem Aufruf des Ressource-Handlers, anhand einer globalen Variable, überprüft, ob die Ressource gerade in Benutzung ist. Ist dies der Fall, wird die Absender-IP verglichen. Stimmt diese mit der Absender-IP der letzten Anfrage überein, kann die Anfrage trotzdem bearbeitet werden. Besteht keine Übereinstimmung, wird die Zeit herangezogen, seit der die Ressource gesperrt ist. Liegt diese mehr als 60 Sekunden zurück, kann die Anfrage ebenfalls bearbeitet werden. Diese Überprüfung ist notwendig, da ein Angreifer die Ressource durch einen einmaligen Aufruf dauerhaft sperren könnte, wenn er seine Anfrage nicht zuende führt.

Bei Bearbeitung der Anfrage, wird die Ressource zunächst generell gesperrt. Ist eine Block-1-Übertragung abgeschlossen, wird die Ressource generell wieder freigegeben. Nur die Bearbeitung einer ClientHello-Nachricht mit korrektem Cookie führt im weiteren Verlauf zu einer erneuten Sperrung, da hier ein Zustand erzeugt wird, der erst in der darauffolgenden Anfrage abgearbeitet wird.

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.

uri syntax [T B05]. beachten wegen session-id

Übertragen werden mit Hilfe der Option ein ClientHello oder ClientKeyExchange + ChangeCipherSpec + Finished, wobei die Länge von Letzterem mit insgesamt 114 (87 + 3 + 22) Byte, durch die einzig definierte Cipher-Suite, fest ist. Anders ist dies bei dem ClientHello, welches durch eine Vielzahl, vom Client beherrschter Cipher-Suites, sehr viel größer werden kann.

Hier erfolgt dann jedoch eine CoAP-Fehlermeldung „4.13 REQUEST ENTITY TOO LARGE“ mit einem Hinweis auf die begrenzte Maximalgröße, so dass ein Client sein Angebot an Cipher-Suites reduzieren kann, um der maximalen Größe gerecht zu werden.

auswertung cookie -> query. cookie in packet ist doof

problem: handshake: uri prüfen !

5.1.6 Update-Funktion

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

5.2 Client

5.3 Testumgebung

flasher überarbeitet. libmc1322x-update

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

dissector in wireshark

Kapitel 6

Evaluation

In den folgenden Abschnitten sollen die Anpassungen von DTLS bewertet werden. Während der Datenverkehr direkt mit dem von DTLS im original verglichen wird, geht es bei der Programmgröße und der Dauer des Handshakes um die Praxistauglichkeit.

6.1 Datenverkehr während des Handshakes

Um einen fairen Vergleich durchzuführen, wird neben dem originalen DTLS auch eine DTLS-Version mit Stateless Header Compression herangezogen. Dazu werden in den folgenden 3 Abschnitten die Datenmengen ermittelt, während im 4. Abschnitt der Vergleich erfolgt. Die Datenmengen basieren auf einem Handshake, bei dem weder Paketverluste noch Angriffe von dritten, oder andere Fehler, auftreten.

Das Umfeld ist in allen Fällen konstant. In einem IEEE 802.15.4 Paket, lassen sich maximal 127 Byte Nutzdaten versenden. Davon werden 48 Byte für den 6LoWPAN-Header, und 8 Byte für den UDP-Header benötigt, so dass 71 Byte pro Paket für den DTLS-Handshake verbleiben. Um den Vergleich auf das Wesentliche zu reduzieren, gehen in den Vergleich zunächst nur die DTLS-Daten selbst ein. Die genannten Header werden dann als „Anzahl der benötigten Pakete“ in der Vergleich mit aufgenommen.

Die in dieser Arbeit genutzte Cipher-Suite dient als Basis für die Ermittlung der Datenmengen. Zu beachten ist hier, dass die dort definierte PRF, keinen Einfluss auf die Datenmenge hat. Für den Handshake sind generell die in Abbildung 6.1 aufgeführten Nachrichten mit den dort angegebenen Größen erforderlich.

Darauf basierende werden nun die Datenmengen für alle 3 Verfahren ermittelt, wobei in Epoche 0, ohne jegliche Verschlüsselung, begonnen wird. In den Abbildungen sind die übertragenen Pakete zur besseren Übersicht jeweils in 3 Gruppen eingeteilt. Jede Gruppe beinhaltet eine vollständige Anfrage des Clients, sowie die vollständige Antwort des Servers.

Typ	Abkürzung	Größe
ClientHello ohne Cookie	CHoC	57 Byte
HelloVerifyRequest	HVR	11 Byte
ClientHello mit Cookie	CHmC	65 Byte
ServerHello	SH	56 Byte
ServerKeyExchange	SKE	87 Byte
ServerHelloDone	SHD	0 Byte
ClientKeyExchange	CKE	87 Byte
ChangeCipherSpec	CCS	1 Byte
Finished	FI	12 Byte

Abbildung 6.1 Größe der Handshake-Nachrichten

6.1.1 DTLS mit Anpassungen

Der Handshake über CoAP, gemäß Abbildung 3.2, erfordert die Übertragung von 30 Datenpaketen, die in Abbildung 6.2 dargestellt sind. Gemäß Abschnitt 3.2 wird eine CoAP-Blockgröße von 32 Byte benutzt. Diese Blockgröße ermöglicht auch dann noch einen Handshake, wenn der DTLS-Header die maximale Größe von 15 Byte annimmt. Zusätzlich zu den übertragenen Handshake-Nachrichten, sind in der letzten Spalte die enthaltenen CoAP-Optionen angegeben. B1 und B2 stehen für die jeweiligen Block-Optionen, während CT die Content-Type-Option beschreibt.

Der Record-Header nimmt hier generell drei Byte in Anspruch. Zwei Byte beinhalten den komprimierten Record-Header, während die Sequenznummer in einem extra Byte angehängt werden muss.

coap packete die nur zur bestätigung dienen und keine daten enthalten rausrechnen zum zusätzlichen vergleich

zu beachten ist, dass 14 Datenpakete mit 116 Byte coap spezifisch sind und keinerlei dtls handshake daten enthalten
die müssen zwar berücksichtig werden, sind aber dennoch eine separate betrachtung wert.

6.1.2 DTLS

maximale payload größe: 127 - 48 - 8 - 13 - 12 - 8 = 38

6.1.3 DTLS mit Stateless Header Compression

mit direkter umsetzung von draft [HB12, Kapitel 3]

Nr.	<->	Record-Header	CoAP-Header	Content-Header	Handshake-Daten	CoAP-Optionen und Inhalt
1	–>	3	13	2	30	URI, B1, CHoC [1/2]
2	<–	3	7			B1
3	–>	3	13		27	URI, B1, CHoC [2/2]
4	<–	3	10	2	11	CT, B1, HVR
5	–>	3	13	2	30	URI, B1, CHmC [1/3]
6	<–	3	7			B1
7	–>	3	13		32	URI, B1, CHmC [2/3]
8	<–	3	7			B1
9	–>	3	13		3	URI, B1, CHmC [3/3]
10	<–	3	4			EMPTY (Separate Antwort)
11	<–	3	11	5	27	CT, B1, B2, (SH, SKE, SHD) [1/5]
12	–>	3	4			EMPTY
13	<–	3	9		32	CT, B2, (SH, SKE, SHD) [2/5]
14	–>	3	4			EMPTY
15	<–	3	9		32	CT, B2, (SH, SKE, SHD) [3/5]
16	–>	3	4			EMPTY
17	<–	3	9		32	CT, B2, (SH, SKE, SHD) [4/5]
18	–>	3	4			EMPTY
19	<–	3	9		20	CT, B2, (SH, SKE, SHD) [5/5]
20	–>	3	4			EMPTY
21	–>	3	22	6	26	URI, B1, (CKE, CCS, FI) [1/4]
22	<–	3	7			B1
23	–>	3	22		32	URI, B1, (CKE, CCS, FI) [2/4]
24	<–	3	7			B1
25	–>	3	22		32	URI, B1, (CKE, CCS, FI) [3/4]
26	<–	3	7			B1
27	–>	3	22		18	URI, B1, (CKE, CCS, FI) [4/4]
28	<–	3	4			EMPTY (Separate Antwort)
29	<–	3	10	4	21	CT, B1, CCS, FI
30	–>	3	4			EMPTY
	<->	90	294	21	405	Gesamt 810

Abbildung 6.2 Datenaustausch während eines Handshake mit angepasstem DTLS

da die angepasste version einen maximalen dtls header ermöglicht, wird hier ebenfalls davon ausgegangen: maximale payload größe: 127 - 48 - 8 - 15 - 14 - 8 = 34

6.1.4 Vergleich

tada

Nr.	<->	Record-Header	Content-Header	Handshake-Daten	Inhalt
1	–>	13	12	38	CHello ohne Cookie [1/2]
2	–>	13	12	19	CHello ohne Cookie [2/2]
3	<–	13	12	11	HelloVerifyRequest
4	–>	13	12	38	CHello mit Cookie [1/2]
5	–>	13	12	27	CHello mit Cookie [2/2]
6	<–	13	12	38	SHello [1/2]
7	<–	13	12	18	SHello [2/2]
8	<–	13	12	38	SKeyExchange [1/3]
9	<–	13	12	38	SKeyExchange [2/3]
10	<–	13	12	11	SKeyExchange [3/3]
11	<–	13	12	0	SHelloDone
12	–>	13	12	38	CKeyExchange [1/3]
13	–>	13	12	38	CKeyExchange [2/3]
14	–>	13	12	11	CKeyExchange [3/3]
15	–>	13	12	1	ChangeCipherSpec
16	–>	13	12	20	Finished inklusive 8 Byte MAC
17	<–	13	12	1	ChangeCipherSpec
18	<–	13	12	20	Finished inklusive 8 Byte MAC
	<->	234	216	405	Gesamt 861

Abbildung 6.3 Datenaustausch während eines Handshake mit DTLS

6.2 Programmgröße

Aktuelle Werte:

Implementierung ist 93897 - 82368 = 11529 = 11,26 KiB groß
8922 Byte setzen sich zusammen:

- 2144 Byte ECC-Funktionen
- 310 Byte AES
- 328 Byte AES-CCM
- 332 Byte AES-CMAC
- 824 Byte Flash-Speicher-Funktionen
- 896 Byte Session-Verwaltung
- 184 Byte Pseudo-Random-Funktion
- 2848 Byte Handshake-Ressource
- 1056 Byte Parse & Send

Gesamtgröße: 93897 = 91,70 KiB
Verbleiben 98304 - 93897 = 4407 = 4,30 KiB

Nr.	<->	Record-Header	Content-Header	Handshake-Daten	Inhalt
1	–>	3	4	34	CHello ohne Cookie [1/2]
2	–>	3	4	23	CHello ohne Cookie [2/2]
3	<–	3	2	11	HelloVerifyRequest
4	–>	3	4	34	CHello mit Cookie [1/2]
5	–>	3	4	31	CHello mit Cookie [2/2]
6	<–	3	4	34	SHello [1/2]
7	<–	3	4	22	SHello [2/2]
8	<–	3	4	34	SKeyExchange [1/3]
9	<–	3	4	34	SKeyExchange [2/3]
10	<–	3	4	19	SKeyExchange [3/3]
11	<–	3	2	0	SHelloDone
12	–>	3	4	34	CKeyExchange [1/3]
13	–>	3	4	34	CKeyExchange [2/3]
14	–>	3	4	19	CKeyExchange [3/3]
15	–>	3	2	1	ChangeCipherSpec
16	–>	3	2	20	Finished inklusive 8 Byte MAC
17	<–	3	2	1	ChangeCipherSpec
18	<–	3	2	20	Finished inklusive 8 Byte MAC
	<->	54	60	405	Gesamt 519

Abbildung 6.4 Datenaustausch während eines Handshake mit DTLS und stateless header compression

Beschreibung	**Basis**	**mit DTLS**
Programm	58760 Byte	70008 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	2048 Byte	2048 Byte
Datensegment	20744 Byte	21025 Byte
Heap	16 Byte	16 Byte
Gesamt	82368 Byte	93897 Byte
	80,44 KiB	91,70 KiB

Abbildung 6.5 Speicheraufteilung von SmartAppContiki ohne und mit DTLS

-> praxistauglich

die maximale stackausnutzung beträgt 1504 byte von 2048
somit wäre es möglich den stack von 2048 Byte auf 1536 Byte zu reduzieren, um 0,5 KiB weiteren Speicher nutzen zu können

Im IETF-Entwurf „A Hitchhiker's Guide to the (D)TLS Protocol“ [TKK13] werden zwar auch Programm-

größen für einzelne Funktionen aufgeführt, die sich jedoch nicht zum direkten Vergleich eignen, da diese für eine 64-bit-Architektur kompiliert wurden.

tinydtls:

aes : code: 1692 byte
var: 4096 byte

sha2 : code: 1072 byte
var: 288 byte

hmac : code: 348 byte

prf : code: 504 byte

6.3 Dauer

- ECC-Multiplikation: ø4,5 s
- Handshake: ø11 s

praxitauglich. trotzdem sinnvoll den handshake durchzuführen, noch bevor anwendungsdaten zum versand anstehen.

Kapitel 7

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
ist alles doof, falls cookie weiterhin in clienthello enthalten -> cookie als query var

contiki speicher weiter optimieren - stacks notwendig ?

Akronyme

6LoWPAN IPv6 over Low power Wireless Personal Area Network, S. 2, 11, 31

ACK Acknowledgement, S. 11
AEAD Authenticated Encryption with Associated Data, S. 4, 17
AES Advanced Encryption Standard, S. 4, 15–18, 20, 23, 24

CBC Cipher Block Chain, S. 4, 23, 24
CCM Counter with CBC-MAC, S. 4, 15–17, 23, 24
CMAC Cipher-based Message Authentication Code, S. 18, 23, 25
CoAP Constrained Application Protocol, S. 1–3, 9–14, 20, 21, 28, 29, 32, 33
CTR Counter, S. 4, 23, 24

DoS Denial-of-Service, S. 7
DTLS Datagram Transport Layer Security Protocol, S. 1–5, 8–15, 17, 19, 21, 22, 26, 31, 32

ECC Elliptic Curve Cryptography, S. 16, 17

HMAC Keyed-Hashing for Message Authentication, S. 16, 18

IETF Internet Engineering Task Force, S. 1–3, 35
IPv6 Internet Protocol, Version 6, S. 2

MAC Message Authentication Code, S. 4, 10, 11, 17, 24, 25

NIST National Institute of Standards and Technology, S. 17

PIN Persönliche Identifikationsnummer, S. 19
PRF Pseudo-Random-Funktion, S. 16–19, 23, 27, 31
PSK Pre-Shared Key, S. 4, 6, 15, 16, 18–20, 27, 28

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

SHA Secure Hash Algorithm, S. 15, 16, 18

SSL Secure Sockets Layer Protocol, S. 1

TCP Transmission Control Protocol, S. 1, 5
TLS Transport Layer Security Protocol, S. 1–3, 5, 6, 8, 10, 13, 14

UDP User Datagram Protocol, S. 1, 5, 6, 10, 11, 13, 31
URI Uniform Resource Identifier, S. 11, 12, 28
UUID Universally Unique Identifier, S. 19

WoT Web of Things, S. 1

Glossar

Cipher-Suite
: Gruppe aus 4 Algorithmen, die für Schlüsselaustausch, Authentifizierung, Hash und Verschlüsselung verwendet werden
S. 2–4, 6, 7, 10, 14–19, 23, 27, 29, 31

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, 5–19, 21, 26–28, 31–35, 47

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. 7, 16, 19

MC13224v
: ARM7TDMI-S Microcontroller der Firma Freescale Semiconductor, Inc
S. 3, 4, 20, 23, 25, 28

RSA-Verfahren
: Asymmetrisches kryptographisches Verfahren zur Verschlüsselung und Signatur, das nach seinen Erfindern Rivest, Shamir und Adleman benannt ist
S. 15

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*.

[Coc09] 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/*.

[DH09] S. Deering und R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460 (Proposed Standard). Internet Engineering Task Force, Dez. 2009. URL: *http://tools.ietf.org/html/rfc2460*.

[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, Dez. 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*.

[Mon+07] G. Montenegro, N. Kushalnagar, J. Hui und D. Culler. *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*. RFC 4944 (Proposed Standard). Internet Engineering Task Force, Sep. 2007. URL: *http://tools.ietf.org/html/rfc4944*.

[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*.

[T B05] L. Masinter T. Berners-Lee R. Fielding. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986 (Proposed Standard). Internet Engineering Task Force, Jan. 2005. URL: *http://tools.ietf.org/html/rfc3986*.

[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

Anhang A

CD und Inhalt

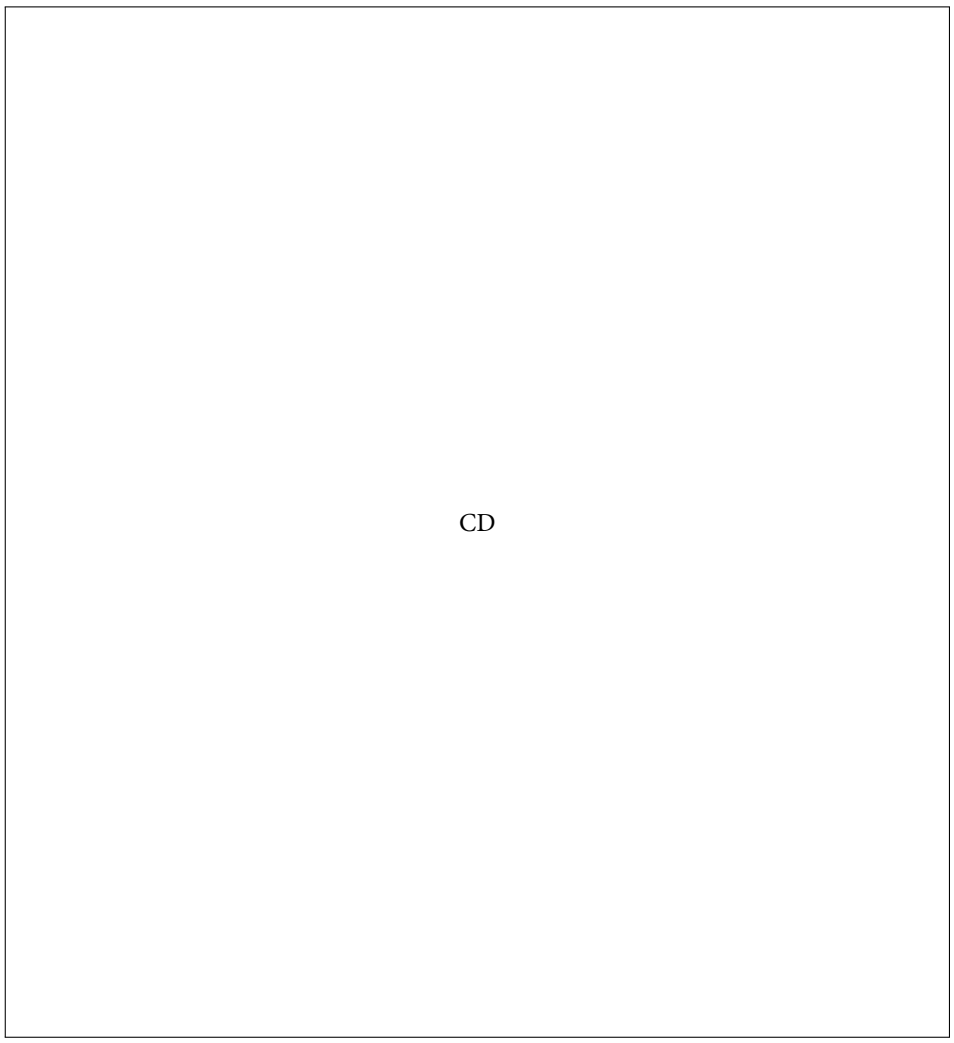

Abbildung A.1 CD

Im Folgenden werden die Ordner der CD, sowie deren Inhalte, in der obersten Ebene beschrieben.

blaster
: todo

border-router
: todo

client
: todo

contiki
: todo

dokumente
: todo

expose
: todo

kolloquien
: todo

libmc1322x
: todo

report
: todo

server
: todo

server-min
: todo

server-tiny
: todo

sniffer
: todo

windows
: todo

wireshark
: todo