Implementierung einer funkbasierten Schlüsselgenerierung für drahtlose Sensornetzwerke

Matthias Schäfer

Bachelorarbeit

Implementierung einer funkbasierten Schlüsselgenerierung für drahtlose Sensornetzwerke

vorgelegt von

Matthias Schäfer

22. Januar 2010

Technische Universität Kaiserslautern
Fachbereich Informatik
AG Distributed Computer Systems Lab

Prüfer:	Prof. Dr. Jens B. Schmitt
weiterer Prüfer:	Prof. Dr. Reinhard Gotzhein
Betreuer:	Dipl-Technoinform. Matthias Stephan Wilhelm

Abstract

Bisherige Implementierungen von Schlüsselaustauschprotokollen für drahtlose Sensornetzwerke kämpfen mit den spezifischen Problemen, die die beschränkten Ressourcen von Sensorknoten mit sich bringen. Auf Grund mangelnden Speichers und hohem Energieverbrauch rechenintensiver Verfahren, sind viele existierende Implementierungen, die einen *ad hoc* Schlüsselaustausch zwischen Sensorknoten unterstützen, für den praktischen Einsatz ungeeignet. Neue Ansätze versuchen ein gemeinsames Geheimnis aus den reziproken Eigenschaften des Mehrwegkanals zwischen zwei Sensorknoten zu extrahieren. In dieser Arbeit wird die Implementierung eines Schlüsselaustauschprotokolls vorgestellt, welches zwei Sensorknoten mit einem 128 bit langen geheimen Schlüssel - extrahiert aus den frequenzselektiven Eigenschaften des Mehrwegskanals - versorgt. Das Protokoll ist für MICAz-Knoten in Form einer wiederverwendbaren TinyOS-Komponente implementiert. Um die praktische Einsetzbarkeit zu analysieren werden die informationstheoretische Sicherheit der Schlüssel und die vom Protokoll beanspruchten Ressourcen getestet. Diese Tests haben eine geschätzte Entropie von bis zu 52 bit pro Schlüssel und eine RAM-Nutzung von 20% auf MICAz-Knoten ergeben. Als Optimierung für den Schlüsselaustausch wird zusätzlich eine grobe Kalibrierung des Received Signal Strength Indicators der Knoten vorgeschlagen und implementiert. Auf diese Weise konnten Schwankungen in den Messungen auf ein Maximum von 1 dB reduziert werden.

Existing implementations of key exchange protocols for wireless sensor networks try to cope with the special challenges of the resource limitations of sensor nodes. Due to lack of memory and high energy consumption of computational expensive algorithms, most of the implementations which provide a key exchange in an ad hoc manner are unusable for actual deployment. Recent efforts in research are made to provide sensor nodes with a shared secret from the reciprocity of the multipath channel between two nodes. In this thesis, a proposed key exchange protocol is implemented which generates a 128 bit secret key out of the frequency-selective, reciprocal channel fading. It was realised in an easy to use TinyOS-component and tested on Crossbow's MICAz sensor nodes. To evaluate the suitability of the implementation, the information-theoretic properties of the generated keys and the resource consumption of the TinyOS-component are analysed. The entropy of the keys is estimated to be 52 bit per key and the RAM usage was bounded to 20% of MICAz's 4 KiB. To enhance the key exchange, a rough calibration of the MICAz's received signal strength indicator (RSSI) is proposed and implemented. Experiments showed a reduction of the variability of two node's RSSI up to ± 1 dB.

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Alle wörtlich oder sinngemäß übernommenen Zitate sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Kaiserslautern, den 22. Januar 2010

Matthias Schäfer

Inhaltsverzeichnis

1 Einleitung

Moderne Szenarien für drahtlose Sensor-Netzwerke (WSN) machen es notwendig Daten vertraulich - d.h. für dritte nicht einsehbar - zu übertragen. Sollen drahtlose Sensorknoten beispielsweise Daten in feindlichem Kriegsgebiet sammeln, ohne dass der Gegner die gleichen Informationen erhält, müssen diese verschlüsselt werden. Symmetrische Verschlüsselungsverfahren verwenden zum Verschlüsseln von Daten ein gemeinsames Geheimnis - einen Schlüssel [28]. Bevor Daten verschlüsselt werden können, muss also erst einmal das *Schlüsselverteilungsproblem* gelöst werden. Die wohl einfachste Lösung - das Verteilen von Schlüsseln im Voraus - ist gefährlich, da im oben genannten Szenario der Gegner physischen Zugang zu den Knoten hat. Auf diese Weise kann der Gegner auf den Speicher der Knoten zugreifen und den dort gespeicherten Schlüssel auslesen [12]. Um sich gegen einen solchen Angriff abzusichern, könnten Schlüssel gruppenweise auf den Sensorknoten verteilt werden, so dass zwei Knoten einen Schlüssel mit einer bestimmten Wahrscheinlichkeit teilen [7]. Dies hätte gegenüber rechenintensiven Schlüsselaustauschverfahren den Vorteil, dass keine komplexen Berechnungen mehr nötig sind und so kostbare Energie gespart werden kann. Auf diese Weise könnte ein Angreifer mit einem aufgedeckten Schlüssel nur einen Teil der Übertragungen entschlüsseln. Dieser Ansatz ist aber aufgrund mangelnden Speichers für die von der WSN-Größe abhängigen Anzahl von Schlüsseln nicht universell einsetzbar. Erschwerend kommt hinzu, dass die Topologie eines WSN im Voraus oftmals nicht bekannt ist und es so im schlimmsten Fall zu einer Partitionierung des WSN's kommen kann [6].

Diese Überlegungen führen zu der Erkenntnis, dass Schlüssel *ad hoc* und bei Bedarf dynamisch erzeugt werden müssen. Hierzu gibt es eine Vielzahl an Schlüsselaustauschprotokollen die sicherstellen sollen, dass zwei Parteien ein gemeinsames Geheimnis über einen unsicheren Kanal erzeugen können. Die Sicherheit vieler Vertreter dieser Protokollgruppe hängt stark von der Komplexität bestimmter mathematischer Probleme ab. Ein bekanntes Verfahren ist der Diffie-Hellman-Schlüsselaustausch. Seine Sicherheit basiert auf der Komplexität der Berechnung des diskreten Logarithmus [24]. Die sehr eingeschränkte Rechenleistung von Sensorknoten macht solche Verfahren aber weitgehend ungeeignet, da sie für den Schlüsselaustausch viele Rechenoperationen durchführen müssen.

Die Sicherheit des von Wilhelm in [37] vorgestellten Verfahrens basiert dagegen auf der Zufälligkeit bzw. der *Unvorhersagbarkeit der Eigenschaften des Funkkanals* zwischen Sensorknoten. Der entscheidende Vorteil dieser Methode für WSNs ist, dass den re-

lativ leistungsschwachen Knoten komplexe Berechnungen erspart bleiben, da die Eigenschaften von Radiowellen von Natur aus komplex sind.

1.1 Ziele dieser Arbeit

Das Hauptziel dieser Arbeit ist die erfolgreiche Implementierung des in [37] vorgestellten Schlüsselaustauschprotokolls als TinyOS-Komponente. Die Implementierung soll erlauben, das Protokoll zur Schlüsselgenerierung universell und einfach zu verwenden, um es so verschiedenen Anwendungsszenarien zugänglich zu machen. Weiter sollen - um die praktische Einsetzbarkeit zu zeigen - Eigenschaften dieses Protokolls evaluiert werden und Designentscheidungen der Implementierung hinterfragt und sinnvoll begründet werden.

Um diese Ziele zu erreichen, werde ich folgendermaßen vorgehen. Kapitel 2 gibt einen groben Überblick über die nötigen technischen Grundlagen und Werkzeuge der Implementierung. Danach wird in Kapitel 3 das Protkoll vorgestellt und im Detail auf die Implementierung eingegangen. Überlegungen und Tests für eine Verbesserung des Protokolls werden in Kapitel 4 angestellt. In der Evaluierung in Kapitel 5 wird das Ergebnis analysiert und die Erreichung der oben genannten Ziele überprüft.

1.2 Verwandte Arbeiten

Es wird viel Aufwand betrieben das Schlüsselverteilungsproblem auf die speziellen Anforderungen von WSNs anzupassen. So haben Watro et al. [36] mit TinyPK die *Low-Exponent*-Variante von RSA und eine eingeschränkte Version des Diffie-Hellman-Verfahrens als TinyOS Komponente implementiert, um vorwiegend eine sichere Kommunikation mit externen Geräten zu gewährleisten. Perrig et al. [25] stellen einen „Sicherheitsbaukasten“ vor, der zum Schlüsselaustausch zwischen zwei Knoten ein Protokoll bietet, welches eine Basisstation als *Trusted Agent* für das Generieren der Schlüssel verwendet.

Das von Liu und Ning [21] vorgestellte TinyECC stellt eine TinyOS-Bibliothek für Elliptic Curve Cryptography (ECC) in WSNs dar. Wie aber Hämäläinen et al. [11] zeigt, können symmetrische Verschlüsselungsverfahren auf typischen Sensorknoten deutlich effizienter berechnet werden als ECC-Algorithmen. Dennoch haben Szczechowiak et al. mit NanoECC [34] eine relativ effiziente Implementierung von ECC für Sensorknoten geschaffen.

Die genannten Protokolle bedienen sich alle den bei leistungsfähigeren Computern eingesetzten mathematischen Methoden zum Schlüsselaustausch. Im Gegensatz dazu bedienen sich Mathur et al. [23] der Zufälligkeit des Mehrwegkanals, welche durch Bewegung der Endknoten verstärkt wird. Das Prinzip der *Radio-Telepathy* ermöglicht

es, durch schnelles gegenseitiges Messen der Eigenschaften des Funkkanals, eine gemeinsame geheime Bitfolge zu generieren. Der Nachteil dieses Protokolls ist, dass die große Anzahl an Messungen sehr schnell geschehen muss und die Parteien sich bewegen müssen. Die Bitrate ist dabei relativ gering, da für ein Bit viele Messungen durchgeführt werden. Azimi-Sadjadi et al. [5] generieren auf ähnliche Weise zufällige Bitfolgen für zwei Endknoten. Statt zweier Schwellwerte für die Abbildung der Signalstärken auf 0 oder 1, wie dies bei der Radio-Telepathy der Fall ist, wird hier nur ein Schwellwert verwendet um sogenannte *Deep Fades* - starke destruktive Interferenzen - zu erkennen und diese auf 0 abzubilden. Bei diesem Verfahren kann aber keine hohe Sicherheit gewährleistet werden, da die meisten Teile der resultierenden Bitfolge leicht zu erraten sind.

Patwari et al. [17] stellen sich allgemein die Frage nach der Eignung der Signalstärke zum Generieren eines geheimen Schlüssels. Sie betrachten dabei statische und dynamische Umgebungen und kommen zu dem Schluss, dass Umgebungen ohne Bewegung ungeeignet sind, um Schlüssel aus den Kanaleigenschaften zu extrahieren. Sie betrachten dabei aber nicht die Frequenzabhängigkeit der Mehrwegausbreitung und die mit ihr verbundene Möglichkeit, mehr Entropie aus einer Position zu ziehen.

2 Grundlagen

Dieses Kapitel bietet eine kurze Übersicht über die verwendete Hard- und Software. Es zeigt die hier relevanten Eigenschaften der verwendeten MICAz Sensorknoten auf und bietet eine kurze Übersicht über das TinyOS-Betriebssystem für drahtlose Sensornetzwerke.

2.1 MICAz-Sensorknoten

Abbildung 2.1: Crossbow's MICAz Sensorknoten

Das Schlüsselaustauschprotokoll wurde implementiert und getestet auf MICAz Sensorknoten der Firma Crossbow Technology Inc., welcher in Abbildung 2.1 dargestellt ist. Diese Knoten sind mit einem Atmel ATmega128L Microcontroller mit 128 KiB Flash-Speicher für Programme ausgestattet. Um Mess- und Kalibrierungsdaten zu speichern ist ein 512 KiB großer Flash-Speicher verfügbar. Zur drahtlosen Übertragung ist ein CC2420 Transceiver Chip von Texas Intruments auf der Plattform integriert [15].

Der CC2420-Transceiver unterstützt den IEEE 802.15.4 Standard, welcher für LR-WPANs[1] auf dem lizenzfreien 2.4 GHz ISM-Band entwickelt wurde. Die Übertragungsrate beträgt dabei 250 kbit/s und es kann insgesamt auf 16 Kanälen (2405-2480 MHz) mit einem Abstand von jeweils 5 MHz übertragen werden. Die Sendeleistung kann in 8 Schritten von -25 dBm bis 0 dBm reguliert werden [14]. Der Chip

[1] Low-Rate Wireless Personal Area Networks

bietet einen *Link Quality Indicator* (LQI) und einen *Received Signal Strength Indicator* (RSSI), um Informationen über die Qualität bzw. die wahrgenommene Leistung einer Übertragung zu bekommen.

Zur Fehlersuche und Ausgabe bietet der MICAz Sensorknoten drei LEDs. Weiter können Daten über einen UART an die serielle Schnittstelle eines PC zur weiteren Verwendung gesendet werden.

MICAz-Sensorknoten werden zum Programmieren über ein sogenanntes *Programming Board* an den PC angeschlossen. In dieser Arbeit wurde Crossbow's MIB510 Programming Board verwendet. Auch die Kommunikation zwischen Knoten und PC erfolgt über dieses Board (siehe Abschnitt 2.2.2).

2.2 TinyOS und nesC

Das Protokoll wurde für das TinyOS-Betriebsystem implementiert. TinyOS ist ein flexibles Betriebssystem, welches für Sensor Netzwerke entwickelt wurde. TinyOS ist plattformunabhängig und kann somit auf verschiedener Hardware (z.B. MICAz, Mica2, Telosb) verwendet werden. Es bietet ein komponentenbasiertes Programmiermodell und unterstützt ereignisgesteuerte Nebenläufigkeit. Wie TinyOS selbst werden TinyOS-Anwendung in der Programmiersprache nesC implementiert [19]. nesC ist eine Erweiterung der Programmiersprache C um die Gliederungskonzepte und das Ausführungsmodell von TinyOS [9]. Ein nesC-Programm ist eine Zusammensetzung aus Komponenten, welche über Schnittstellen, die sie bereitstellen oder benutzen, verbunden sind. Schnittstellen bestehen bei nesC aus *Commands* und *Events*. Commands dienen dazu, eine Operation zu starten und Events signalisieren die Terminierung einer Operation. Operationen nach diesem Modell werden *Split-Phase-Operationen* genannt [8]. Schnittstellen können von mehreren Komponenten (z.B. für verschiedene Plattformen) implementiert werden.

Die Komponenten werden in TinyOS in zwei Arten unterteilt. Es gibt zum einen die *Module*, welche die Implementierung einer Schnittstelle oder einer Applikation enthalten. Zum anderen gibt es *Konfigurationen*, in welchen Schnittstellen mit Modulen bzw. anderen Konfigurationen verknüpft werden.

Zur Identifizierung von Sensorknoten dient die von TinyOS beim Kompilieren festgelegte, 16 bit breite Variable `TOS_NODE_ID`. Zum Kompilieren und Installieren von Anwendungen auf Sensorknoten bietet TinyOS für jede Plattform Makefiles für das Werkzeug `make` [1]. Für jede TinyOS-Anwendung wird ebenfalls ein Makefile erstellt, welches das mitgelieferte Makefile einbindet und die Hauptkomponente der Anwendung identifiziert. Ein minimales Beispiel für ein Makefile für eine Anwendung `TestAppC` könnte folgendermaßen aussehen:

```
COMPONENT=TestAppC
include $(MAKERULES)
```

Um beispielsweise einen MICAz-Sensorknoten über ein MIB510 Programming-Board mit dieser Anwendung zu programmieren, wäre folgender Befehl nötig:

```
make micaz install,1 mib510,/dev/ttyS0
```

Die Zahl 1 als Argument von `install` gibt die `TOS_NODE_ID` des Knoten an. Der Pfad am Ende des Befehls gibt den betriebssystemabhängigen Geräteknoten an, der den seriellen Anschluss zum Programming-Board repräsentiert.

2.2.1 CC2420/Drahtlose Kommunikation

Für diese Arbeit ist vor allem eine umfassende Unterstützung drahtloser Kommunikation und ein guter Zugang zu Informationen über den Funkkanal wichtig. Zu diesem Zweck fanden folgende Schnittstellen Verwendung:

`CC2420Config` ist eine Hardwareabstraktionsschicht (HAL), welche das Einstellen des Transceiver Chips ermöglicht. Parameter wie der verwendete Übertragungskanal können verändert werden und mit dem `sync()`-Command werden Änderungen von der Hardware übernommen.

`CC2420Packet` bietet Zugriff auf den plattformabhängigen Teil der von TinyOS grundlegenden Netzwerkabstraktion, den sog. *Active Messages* (AM). Diese sind single-hop Pakete mit Attributen wie maximaler Größe, Empfängeradresse und einem Typ, welcher das Protokoll identifiziert. Das in dieser Arbeit implementierte Schlüsselaustauschprotokoll beispielsweise hat den AM-Typ 6. `CC2420Packet` erlaubt es die Sendeleistung (TX-Power) für einzelne Pakete festzulegen und die oben genannten LQI- und RSSI-Werte eines empfangenen Pakets auszulesen.

`AMPacket` erlaubt es auf die plattformunabhängigen Felder von AMs wie Empfänger- und Senderadresse oder den AM-Typ zuzugreifen.

`AMSend` und `AMReceive` definieren Commands zum Senden und Events zum Empfangen von Active Messages.

2.2.2 Serielle Kommunikation

Zum Analysieren des in dieser Arbeit implementierten Protokolls werden Daten über dessen Ablauf benötigt. TinyOS bietet die Möglichkeit, Daten von den Knoten zu einem PC über die serielle Anbindung des *Programming-Boards* (z.B. Crossbow's

MIB510) zu senden. Für diese serielle Kommunikation wird - wie bei der drahtlosen Kommunikation - die AM-Schicht von TinyOS benutzt. Auf diese Weise können für die Kommunikation zwischen Knoten und PC die selben Schnittstellen benutzt werden, die auch für die drahtlose Kommunikation zwischen zwei Knoten verwendet werden. Die Daten können dann vom PC gelesen, interpretiert und weiterverarbeitet werden. TinyOS bietet hierfür untere anderem folgende Java-Klassen und Werkzeuge:

`net.tinyos.message` ist ein Java-Paket, welches das Empfangen und Senden von Nachrichten über den seriellen Anschluss ermöglicht. Es bietet weiter Klassen zum Kodieren, Dekodieren und Modifizieren von TinyOS-Nachrichten.

`mig` (**M**essage **I**nterface **G**enerator) ist ein Werkzeug, um Code zum Verarbeiten von TinyOS-Nachrichten zu erzeugen. mig generiert unter anderem Java-Klassen, die Nachrichten repräsentieren und es ermöglichen auf die einzelnen Felder der Nachrichten zuzugreifen, indem es die Definitionen aus nesC in andere Sprachen übersetzt.

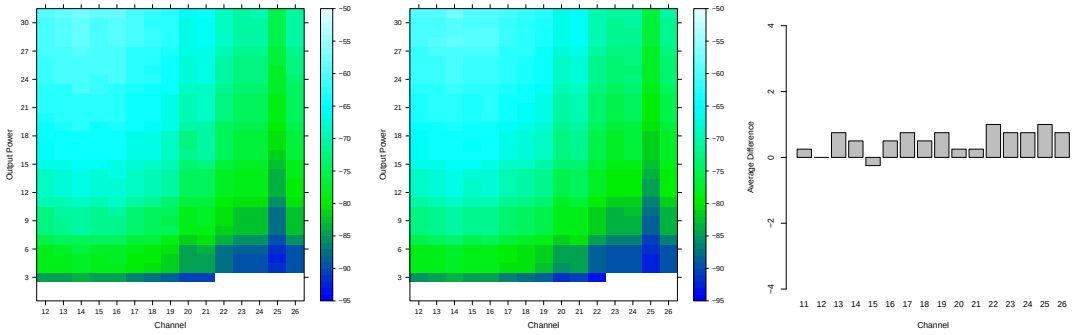
3 Schlüsselgenerierung

Dieses Kapitel stellt die Implementierung im Detail vor. Dabei wird von den übertragungstechnischen Grundlagen ausgehend zuerst der Hintergrund und das Konzept des Protokolls beschrieben. Somit sind alle Voraussetzungen für die Implementierung geschaffen und es kann näher auf die praktische Umsetzung eingegangen werden.

3.1 Konzept

Der Funkkanal zwischen einem Sender und einem Empfänger wird von vielen Faktoren beeinflusst. Das Signal kommt beim Empfänger nicht mehr in seiner ursprünglichen Form an. Zum einen verliert das Signal auf seinem Weg zum Empfänger durch *Dämpfung* an Stärke. Berücksichtigt man sonst keine Effekte und kann sich das Signal ungehindert ausbreiten, ist die empfangene Signalstärke $P_r \approx P_0/r^2$. Dabei beschreibt P_0 die ursprüngliche Sendestärke und r den Abstand zwischen Sender und Empfänger. Zum anderen hängt die vom Empfänger wahrgenommene Signalstärke stark von der Umgebung ab. So werden elektromagnetische Wellen durch *Abschattung* aufgrund großer Objekte stark gedämpft. Auch *Reflexionen* an Objekten nehmen großen Einfluss auf die Signalausbreitung. Weitere Effekte sind *Streuung*, bei welcher das Signal an Objekten in mehrere schwächere Signale aufgespalten wird, eine *Beugung* des Signals an Objektkanten und eine *Brechung* des Signals bei einer Dichteänderung des Mediums. Das Zusammenwirken der genannten Effekte führt zur *Mehrwegausbreitung*. Sie bewirkt, dass das gleiche Signal auf unterschiedlichen Wegen zum Empfänger gelangt. Dabei kommt es zur sogenannten *Laufzeitdispersion*, d.h. das gleiche Signal hat verschiedene Verzögerungen aufgrund der unterschiedlichen Wege. Auf diese Weise kann das Signal mit sich selbst interferieren, was wiederum zu starken Schwankungen in der empfangenen Signalstärke führen kann. Wie [3] zeigt, hat das Verhalten der empfangenen Signalstärke durch die Mehrwegausbreitung einen chaotischen Charakter und ist auf Grund der vielen Faktoren, die diese beeinflusst, nur sehr schwer vorherzuberechnen [29].

Das Auftreten und das Ausmaß der genannten Effekte sind von der Wellenlänge abhängig. Da die Wellenlänge λ durch $\lambda = c/f$ gegeben ist, wobei c die Ausbreitungsgeschwindigkeit und f die Frequenz ist, ist sie direkt von der Frequenz und somit



(a) Sicht Knoten A (b) Sicht Knoten B (c) Unterschiede

Abbildung 3.1: Beispiel für die Reziprozität eines Kanals. Die beiden Signal-Maps (links) zeigen den Kanal zwischen zwei Knoten (A und B) und das rechte Balkendiagramm zeigt die durchschnittlichen Unterschiede in der empfangenen Signalstärke (in dBm) für die stärkste Sendeleistung.

vom Übertragungskanal des Sensorknoten (vgl. Abschnitt 2.1) abhängig. Die Auswirkungen der Effekte unterscheiden sich damit von Übertragungskanal zu Übertragungskanal.

Abbildung 3.1a zeigt eine Visualisierung der Effekte. Hier wurden Daten von Knoten A zu Knoten B mit verschiedenen Sendestärken auf verschiedenen Frequenzen gesendet und dabei die empfangene Signalstärke gemessen. Auf der Abszissenachse wurde dabei der Kanal (Frequenz) abgetragen und auf der Ordinatenachse die Sendeleistung. Die Farbe repräsentiert die empfangene Signalstärke in dBm. Es ist deutlich zu erkennen, dass die Signalstärke beim Empfänger von der Frequenz bzw. dem Übertragungskanal abhängt.

Die entgegengesetzte Richtung ist in Abbildung 3.1b zu sehen. Hier wurden die Rollen von Sender und Empfänger vertauscht. Vergleicht man die beiden Grafiken, stellt man eine starke Ähnlichkeit fest. Diese Ähnlichkeit der Sichten auf den Kanal zwischen den beiden Knoten basiert auf dem *Prinzip der Reziprozität*. Dieses Prinzip besagt, dass zwei Transceiver die gleichen Kanaleigenschaften wahrnehmen, wenn die Rolle des Senders und des Empfängers vertauscht werden, sofern sich die Ausbreitungseigenschaften des Signals nicht ändern [37]. Abbildung 3.1c verdeutlicht nochmals die Ähnlichkeit des wahrgenommenen Kanals bei beiden Knoten für die stärkste Sendeleistung, indem die Differenzen der empfangenen Signalstärken betrachtet werden. In diesem Beispiel beträgt die maximale Differenz 1 dBm auf den Kanälen 22 und 25.

Durch die hohe Komplexität der Signaleigenschaften, die Frequenzabhängigkeit der Effekte und der Reziprozität des Kanals teilen zwei Knoten ein gemeinsames Geheimnis: Die empfangene Signalstärke auf den verschiedenen Frequenzen.

Wie Experimente in [37] gezeigt haben, hat schon eine kleine Positionsänderung eines Sensorknotens eine große Auswirkung auf das Signalbild des Kanals. Somit müsste ein Außenstehender die Kanaleigenschaften für die Positionen zweier Knoten vorhersagen um das Geheimnis zu kennen. Die Sicherheit des Geheimnisses basiert also auf der Unvorhersagbarkeit der Kanaleigenschaften für eine bestimmte Position.

Aus diesen Überlegungen heraus hat Wilhelm in [37] ein Protokoll entworfen, welches es einem Sensorknotenpaar ermöglicht, das geteilte Geheimnis zu extrahieren und daraus einen Schlüssel zu generieren.

3.2 Protokoll

Das Schlüsselaustauschprotokoll besteht aus drei Phasen. Die erste Phase - die sogenannte *Sampling-Phase* - ist dafür zuständig Messungen für verschiedene Frequenzen durchzuführen und so die Signaleigenschaften des Funkkanals herauszufinden. In der darauf folgenden *Key Generation Phase* werden Abweichungen in den Messungen mit Fehlerkorrekturverfahren ausgeglichen und aus den daraus resultierenden Informationen ein geheimer Schlüssel generiert. In der dritten Phase - der *Acceptance Phase* - wird sichergestellt, dass beide Parteien das gleiche Geheimnis erzeugt haben. Protokoll 1 auf Seite 12 zeigt das gesamte Protokoll in Pseudocode. Im folgenden werde ich in Anlehnung an die Literatur über Sicherheit bei drahtloser Kommunikation die Parteien als Alice und Bob und den feindlichen Lauscher als Eve bezeichnen. Eve soll hier im Stande sein alle Nachrichten, die zwischen Alice und Bob ausgetauscht werden, mitzuhören.

3.2.1 Sampling Phase

Diese erste Phase des Protokolls dient dazu, die Signalstärken auf allen m Übertragungskanälen abzuschätzen. Hierzu werden für jeden Kanal n Messungen der empfangenen Signalstärke durchgeführt, indem sogenannte *Sample*-Nachrichten ausgetauscht werden. Der Durchschnitt aller gesammelten Messungen eines Kanals wird als Zustand dieses Kanals gespeichert, bis für jeden Kanal der Zustand μ_k ($1 \leq k \leq m$) geschätzt wurde. Der Vektor $\mu = (\mu_1, \ldots, \mu_m)$ enthält am Ende dieser Phase den Zustand des gesamten Übertragungskanals aus der Sicht des Knoten. War das Kanal-Sampling erfolgreich, sind die Zustandsvektoren μ_A von Alice und μ_B von Bob gleich. Da der drahtlose Funkkanal aber nicht symmetrisch ist und es Schwankungen aufgrund der Hardware gibt, kommt es zu Abweichungen $d = |\mu_A - \mu_B| \geq 0$ (vgl. Abbildung 3.1c). Um trotz dieser Abweichungen das gemeinsame Geheimnis aus den Informationen zu extrahieren, wird in der nächsten Phase eine Fehlerkorrektur durchgeführt.

```
1. Sampling Phase:
  for all c ∈ CHANNELS do
    // n - Anzahl der Samples
    for i = 1 to n do
      // messe Signalstärke beim Empfang
      rss[i] = exchangeSampleOnChannel(c);
    end for
    rss_mean[c] = calcMean(rss);
  end for

2. Key Generation Phase:
  if amIAlice() then
    for all c ∈ CHANNELS do
      tolerances[c] = chooseTolerance(c);
      code[c] = correctError(rss_mean[c], tolerances[c]);
    end for
    recs = calcReconciliation(rss_mean, code);
    send(tolerances, recs);
  else
    receive(tolerances, recs);
    code = repairMeasurements(rss_mean, tolerances, recs);
  end if
  seed = concat(bin(code));

3. Acceptance Phase:
  hash = hashFunction(seed);
  remote_hash = exchange(hash);
  if remote_hash == hash then
    acceptSecret(seed);
  else
    recalcTolerances();
    redo(Key Generation Phase);
  end if
```

Protokoll 1: Schlüsselaustauschprotokoll

3.2.2 Key Generation Phase

In dieser Phase wird aus dem gemessenen Zustand μ_k des Kanals k zwischen Alice und Bob ein passender Code erzeugt. Damit der Code als gemeinsames Geheimnis verwendet werden kann, müssen die Unterschiede in den Messungen korrigiert werden. Dies geschieht durch die Anwendung einer geeigneten Fehlerkorrektur.

Sei $\mathcal{M}$ die Menge aller möglichen Werte die μ_k annehmen kann. Mit der Abstands-

funktion $dis(x,y) = |y - x|$ mit $x, y \in \mathcal{M}$ bildet $\mathcal{M}$ einen metrischen Raum. Dieser metrische Raum ist in diesem Fall durch die Hardware des Sensorknotens gegeben, da sie nur einen bestimmten Bereich $[\mu_{min}, \mu_{max}]$ an Signalstärken messen kann[1]. Zum Korrigieren der Differenz $dis(\mu_A, \mu_B)$ wird für jeden Kanal k eine Toleranz t_k gewählt. Mit dieser Toleranz wird eine Teilmenge $C \subseteq \mathcal{M}$ des metrischen Raums so definiert, dass zu jedem $\mu_k \in \mathcal{M}$ ein $c_k \in C$ existiert, so dass $dis(\mu_k, c_k) \leq t_k$ gilt. Weiter gilt für alle $c_i, c_j \in C$, dass $dis(c_i, c_j) \geq 2 \cdot t$. Dies bedeutet, dass der Abstand zwischen zwei benachbarten Elementen in C genau der zweifachen Toleranz entspricht. Die Funktion $enc : \mathcal{M} \to C$ bildet nun alle $\mu \in \mathcal{M}$ auf das $c \in C$ mit dem geringsten Abstand ab. Dann ist C unser Code und *enc* unsere Kodierung. Mit ihrer Hilfe können *manche* Schwankungen zwischen μ_A und μ_B, deren absoluter Wert kleiner oder gleich der Toleranz sind korrigiert werden.

Beispiel: Sei $\mathcal{M} = [-95, 0] \subset \mathbb{R}$ und $t = 1$. Dann könnte C folgendermaßen aussehen:

$$C = \left\{ -94 + i \cdot (2 \cdot t) \,\middle|\, i = 0, \ldots, floor\left(\frac{|\mu_{max} - \mu_{min}|}{2 \cdot t}\right) \right\} \tag{3.1}$$

$$= \{-94, -92, \ldots, -2, 0\}$$

Angenommen Alice hätte für Kanal 11 eine Signalstärke von $\mu_{A,11} = -76.1\ dBm$ und Bob $\mu_{B,11} = -76.9\ dBm$. Dann würde gelten

$$enc(\mu_{A,11}) = enc(-76.1) = 76 = enc(-76.9) = enc(\mu_{B,11})$$

und die Schwankung wäre somit korrigiert.

Um wirklich alle Schwankungen korrigieren zu können, die kleiner oder gleich der Toleranz sind, muss zusätzlich noch eine Anpassung vorgenommen werden. Angenommen im obigen Beispiel wären $\mu_{A,11} = -76.8$ und $\mu_{B,11} = -77.4$. Dann gilt $enc(\mu_{A,11}) = -76$, für $\mu_{B,11}$ gilt aber $enc(\mu_{B,11}) = enc(-77.4) = -78$. Die beiden Messwerte werden also auf zwei verschiedene Codes abgebildet obwohl ihr Abstand kleiner als die Toleranz ist. Um solchen Fällen entgegenzuwirken, berechnet Alice zusätzlich zum Codewort $enc(\mu_A) = c$ den Wert $d = c - \mu_A$ und sendet diesen an Bob. Dieser kodiert seinerseits nicht den Wert μ_B, sondern den Wert $\mu_B + d$. Auf diese Weise werden *alle* Werte mit einer Schwankung, die kleiner oder gleich der Toleranz ist mit *enc* auf den gleichen Wert abgebildet.

[1] Bsp. MICAz: $\mu_{min} = -95$ und $\mu_{max} = 0\ dBm$ [15, 14]

Beweis. Seien $\mu_A, \mu_B \in \mathcal{M}$ mit $dis(\mu_A, \mu_B) \leq t$, $c = enc(\mu_A)$ und $d = c - \mu_A$.
Dann gilt:

$$\begin{aligned} dis(\mu_A, \mu_B) &= dis(\mu_A + d, \mu_B + d) \\ &= dis(\mu_A + dis(\mu_A, c), \mu_B + d) \\ &= dis(\mu_A + (c - \mu_A), \mu_B + d) \\ &= dis(c, \mu_B + d) \leq t \end{aligned}$$

$$\Rightarrow enc(\mu_B + d) = c$$

□

Nach Anwendung des Fehlerkorrekturverfahrens berechnen Alice und Bob aus den entstandenen korrigierten Zustandsvektoren μ'_A und μ'_B das geheime Codewort, indem sie die Messwerte mit einer vorher vereinbarten Funktion *bin* auf eine Bitfolge $s_A = bin(\mu'_A)$ und $s_B = bin(\mu'_B)$ abbilden.

Mit diesem Verfahren können sich Alice und Bob auf ein gemeinsames Geheimnis einigen, ohne dass Eve durch die preisgegebenen Informationen Rückschlüsse über das Codewort ziehen kann. Sind die Differenzen in den Messwerten aber höher als die verwendeten Toleranzen, haben Alice und Bob unterschiedliche Codewörter generiert. Die Aufgabe der letzten Phase ist es, sicherzustellen, dass Alice und Bob sich auf ein gleiches Codewort geeinigt haben, ohne jedoch Informationen über das Geheimnis preiszugeben.

3.2.3 Acceptance Phase

In der dritten Phase wird das Ergebnis auf beiden Seiten auf Gleichheit überprüft. Um Eve keine Informationen über das gemeinsame Geheimnis zu offenbaren, wird hier ein *Challenge-Response-Verfahren* verwendet. Alice berechnet mit einer Hashfunktion den Hashcode ihres Codeworts und sendet dieses an Bob. Bob tut dies gleich und vergleicht die beiden Hashcodes. Stimmen sie überein haben sich beide Seiten erfolgreich auf ein geheimes Codewort geeinigt. Stimmen die beiden Codewörter nicht überein, versuchen Alice und Bob die Toleranzen so zu erhöhen, dass die Key Generation Phase erfolgreich wiederholt werden kann. Da aber keiner von beiden Parteien beide Zustandsvektoren μ_A und μ_B kennen kann, ohne diese Eve zu offenbaren, muss diese Toleranzerhöhung heuristisch geschehen.

3.3 Sicherheit

Die Sicherheit des Protokolls ist in diesem Kontext die *Unvorhersagbarkeit* des generierten Schlüssels. Das bedeutet, dass Eve trotz Belauschen und Kenntnis des Proto-

kolls sowie der Umgebung nicht in der Lage sein soll vorauszusagen welchen Schlüssel die beiden Parteien generieren. Wie Wilhelm in [37] zeigt, ist das Signalbild des Kanals und somit das geteilte Geheimnis von Position zu Position sehr unterschiedlich. Ist Eve also nicht in nahezu der selben Position wie Alice oder Bob, nimmt sie einen völlig anderen Funkkanal wahr wie Alice und Bob [5]. Durch Belauschen kommt Eve hier offensichtlich an keinerlei Informationen über den Schlüssel selbst. Es werden während der Key Generation Phase zwar Toleranzen und Anpassungsvektoren übertragen, aber diese geben keine Auskunft über das gemessene Geheimnis selbst. Eine mögliche Angriffsstelle bietet die Hash-Funktion in der Acceptance Phase. Aber der Einsatz von z.B. dem MD5-Algorithmus sollte bezüglich Sicherheit unbedenklich sein, da bis jetzt lediglich bzgl. Kollisionsfreiheit eine Schwachstelle bekannt ist und diese hier nicht gefordert wird [27].

Eine andere Möglichkeit die Eigenschaften des Funkkanals zwischen Alice und Bob vorherzusagen wäre Signalausbreitungsmodelle zu verwenden, um die Signalstärke beim Empfänger von außen abzuschätzen. Es gibt eine Vielzahl von Modellen, die auf verschiedenen Techniken wie *Ray-Tracing* basieren oder versuchen unter Annahme von Idealbedingungen die physikalischen Effekte zu berechnen. In manchen Modellen können recht genaue Vorhersagen gemacht werden, wie z.B. in [4]. Es wird versucht mit Hilfe eines dreidimensionalen Ray-Tracing-Modells die RSS für eine bestimmte Position zu berechnen. Für NLOS[2]-Regionen nahe des Transmitters konnten hier Vorhersagen mit einem Fehler, der kleiner als 3 dB war, gemacht werden. Allerdings fanden die Experimente unter Laborbedingungen statt. In einer realen - komplexen - Umgebung geht der Rechenaufwand solcher Modelle über die Fähigkeiten von aktuellen Rechnern hinaus. Weiter muss die Menge an Eingabeparametern sehr hoch sein um genaue Berechnungen durchführen zu können. Es wird Wissen über die Umgebung benötigt, welches im Normalfall nicht vorhanden ist. Andere Arten von Modellen sind zwar effizienter, können aber nur sehr eingeschränkt angewendet werden oder sind schlichtweg unbrauchbar [16].

Unter der Annahme, dass Eve die genaue Verteilung der RSS-Werte zwischen Alice und Bob kennt, könnte diese die Werte mit einer gewissen Wahrscheinlichkeit erraten. Mit einer *Brute-Force-Methode* z.B. könnte Eve den Schlüssel mit einer bestimmten Wahrscheinlichkeit mit akzeptablem Aufwand herausfinden. Wie hoch diese Wahrscheinlichkeit ist, hängt direkt von der Anzahl von Möglichkeiten für Codewörter ab, welche wiederum direkt von den verwendeten Toleranzen abhängt. Je niedriger die Toleranz, desto höher die Anzahl und desto niedriger die Wahrscheinlichkeit ein bestimmtes Codewort zu finden. Angenommen die RSS-Werte - und somit die Codewörter - wären zur starken Vereinfachung gleichverteilt. Dann müsste die Brute-Force-Suche alle möglichen Kombinationen für Messwerte überprüfen. Bei einer Toleranz von $t = 1$ für alle $m = 16$ Kanäle, einer minimalen RSS von $rss_{min} = -95\,\text{dBm}$

[2]Non Line Of Sight - keine direkte Sichtverbindung

```
#include "KeyGen.h"

interface KeyGen {

        /**
         * This function starts the key exchange protocol with
         * remote node node_id.
         */
        command void generateKey (uint16_t node_id);

        /**
         * This event get invoked when the key generation finished.
         */
        event void generationDone (secret_key *key, error_t error);

}
```

Listing 3.1: KeyGen Interface

und einer maximalen RSS von $rss_{max} = 0\,\text{dBm}$, müsste Eve

$$\underbrace{\left\lceil \frac{rss_{max} - rss_{min}}{2 \cdot t} \right\rceil^{m}}_{\text{\# möglicher Codewörter}} = 48^{16} \approx 7,94 \cdot 10^{26}$$

Möglichkeiten durchprobieren. Dies würde einer Trefferwahrscheinlichkeit von ca. $1,26 \cdot 10^{-25}\%$ pro Versuch entsprechen. Bei einer Toleranz von 10 dagegen wären es nur noch ungefähr $152,59 \cdot 10^{9}$ Möglichkeiten. Man sieht, dass die Trefferwahrscheinlichkeit stark von der verwendeten Toleranz abhängt. Weiter muss beachtet werden, dass die RSS-Werte in der Realität nicht gleichverteilt sind. Wie Experimente in [37] gezeigt haben, ist die Verteilung der RSS-Werte ähnlich einer Rayleighverteilung. Unter genauer Kenntnis der Verteilung zwischen Alice und Bob, könnte Eve mit einer selektiven Brute-Force-Suche die Trefferwahrscheinlichkeit erhöhen.

Weitere Überlegungen und eine Analyse der Sicherheit der folgenden Implementierung sind in Kapitel 5.1.2 zu finden.

3.4 Implementierung

Die Implementierung des Protokolls in nesC erfolgt in mehreren TinyOS Komponenten. Das eigentliche Protokoll ist in dem Modul `KeyGenP` implementiert. Die dazugehörige Konfiguration `KeyGenC` verbindet alle von `KeyGenP` verwendeten Interfaces mit den dazugehörigen Modulen. Um die Komplexität des Protokolls vor dem Benutzer

zu verstecken und um die Komponente wiederverwendbar zu machen, stellen beide Komponenten das in Listing 3.1 abgebildete Interface bereit. Die Komponenten und das Interface folgen dem allgemeinen Software Design Muster von TinyOS [8].

Das Interface bietet zur Schlüsselgenerierung mit dem vorgestellten Protokoll eine Split-Phase-Operation. Diese besteht aus dem Command `generateKey`, welches den Schlüsselaustausch initiiert. Dieses Command bekommt als Argument den Sensorknoten-Bezeichner, der denjenigen Knoten identifiziert, mit welchem der Schlüsselaustausch stattfinden soll (siehe Abschnitt 2.2). Damit das Schlüsselaustauschprotokoll funktionieren kann, benötigen die Sensorknoten im WSN demnach eindeutige Bezeichner. Weiter muss die Operation auf beiden Knoten ausgeführt werden. Der Zeitpunkt für das beidseitige Ausführen des `generateKey`-Commands wird von der jeweiligen Anwendung festgelegt.

Das zum `generateKey`-Command gehörende Event `generationDone` signalisiert die Terminierung des Schlüsselaustauschprotokolls. Dieses Event muss von der Komponente, die das `KeyGen`-Interface benutzt, implementiert werden und bekommt zwei Argumente:

1. `key` ist ein Zeiger, welcher auf eine `secret_key`-Datenstruktur zeigt. Diese Datenstruktur enthält - sofern der Schlüsselaustausch erfolgreich war - den generierten Schlüssel und einen *Security Indicator* (SI). Der $\text{SI} \in [0,1] \subset \mathbb{R}$ gibt Auskunft über die verwendeten Toleranzen und die damit verbundene Sicherheit des Schlüssels (vgl. Abschnitt 3.3). Details über die Berechnung des SI und des Schlüssels sind in Abschnitt 3.4.1 zu finden.

   Die Datenstruktur `secret_key` ist in der zum Interface gehörenden Header-Datei `KeyGen.h` folgendermaßen definiert:

   ```
   typedef nx_struct secret_key {
           nx_uint8_t secret[NUM_OF_CHANNELS];
           nx_float si; // security indicator
   } secret_key;
   ```

   Die Konstante `NUM_OF_CHANNELS` gibt die Anzahl an Kanälen an und ist im Bezug auf den CC2420-Chip auf den Standardwert 16 gesetzt. Daraus resultiert also ein $16 \cdot 8 = 128$ bit langer Schlüssel.

2. Der zweite Parameter `error` vom TinyOS-Typ `error_t` gibt an, ob der Schlüssel erfolgreich generiert wurde. Gilt `error` $=$ `SUCCESS`, so ist das Protokoll erfolgreich terminiert. Kommt es zu einem Fehler, wird dies durch `error` $=$ `FAIL` signalisiert und `key` ist der Nullzeiger und somit ungültig.

Das `KeyGen`-Interface und die `KeyGenC`-Komponente können für beliebige Anwendungen verwendet werden. Dazu muss lediglich das Interface benutzt und in der Hauptkomponente der Anwendung mit der `KeyGenC`-Komponente verbunden werden. Wann, mit wem und für welchen Zweck ein Schlüsselaustausch stattfinden soll,

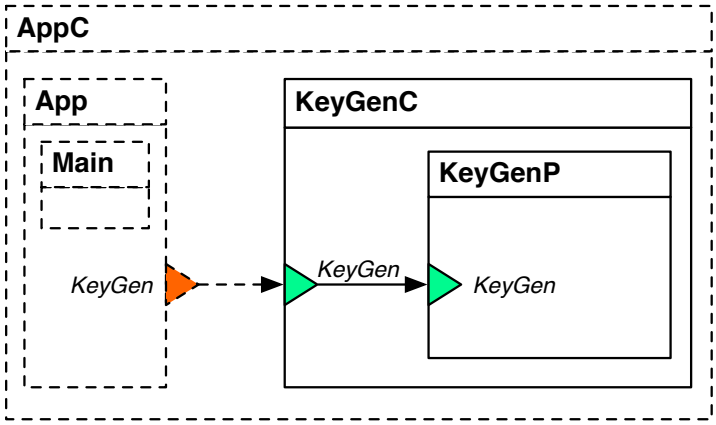
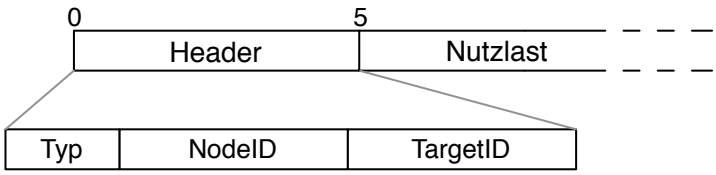


Abbildung 3.2: Schematische Beispielzusammensetzung einer TinyOS-Anwendung, die das Schlüsselaustauschprotokoll verwendet. Dreiecke, die in eine Komponente zeigen, repräsentieren Schnittstellen die von dieser Komponente bereitgestellt werden. Dreiecke, die aus einer Komponente zeigen, repräsentieren Schnittstellen die von dieser Komponente verwendet werden.

kann von der Anwendung frei entschieden werden. Ein Beispiel für die Einbindung und die Zusammensetzung des Schlüsselaustauschprotokolls und seiner Komponenten ist in Abbildung 3.2 schematisch dargestellt.

Für die Kommunikation zwischen den Knoten während des Schlüsselaustauschs werden Nachrichten verschiedener Typen ausgetauscht. Zu diesem Zweck sind in der `KeyGen.h`-Datei mehrere Nachrichtenformate und Typen definiert. Ohne Berücksichtigung der von den in Abschnitt 2.2.1 erwähnten TinyOS Kommunikationsschichten angehängten Metadaten, haben alle vom Protokoll ausgetauschten Nachrichten folgendes Format:



Sie beginnen mit einem 5 Bytes großem Nachrichtenkopf, der den Nachrichtentyp (1 Byte), die Absender- und die Empfänger-ID (jeweils 2 Bytes) enthält. Ein Knoten verarbeitet wärend dem Protokoll eine Nachricht nur dann, wenn sie an ihn selbst adressiert ist, d.h. wenn `TargetID` = `TOS_NODE_ID` *und* `NodeID` = `remote_node`[3] gilt. Im Folgenden werde ich dieses Wissen voraussetzen und bei der Verwendung der

[3] `remote_node` entspricht dem Knotenbezeichner, welcher beim Aufruf von `generateKey` übergeben wurde

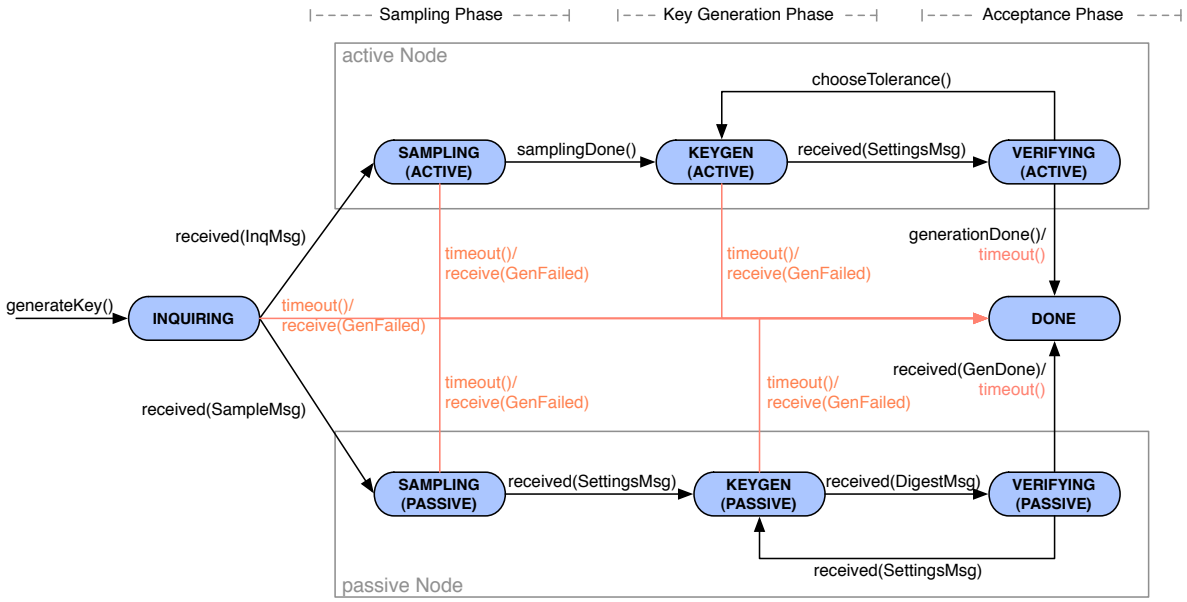


Abbildung 3.3: Zustandsübergangsdiagramm für einen Knoten während des Schlüsselaustauschprotokolls

Begriffe *„empfangen“* bzw. *„senden“* davon ausgehen, dass sie an den betreffenden Knoten adressiert sind. Nach dem Kopf folgt die vom Nachrichtentyp abhängige Nutzlast. Das Protokoll kennt folgende in der `KeyGen.h`-Datei definierten Nachrichtentypen:

→ Typ 0: `InquiryMsg`

→ Typ 1: `SampleMsg`

→ Typ 2: `SettingsMsg`

→ Typ 3: `VerificationMsg`

→ Typ 4: `GenerationDoneMsg`

Nähere Beschreibungen der einzelnen Nachrichtentypen und deren Aufgabe werden im Folgenden vorgestellt.

3.4.1 Protokoll

Die Implementierung des Protokolls, die sich hinter dem `KeyGen`-Interface verbirgt, liegt in der `KeyGenP`-Komponente. Diese implementiert zusätzlich zu den drei Phasen des Schlüsselaustauschprotokolls noch eine vierte Phase, die *Inquiry Phase*. Ihre Aufgabe ist es alles nötige vorzubereiten, damit das Schlüsselaustauschprotokoll problemlos ausgeführt werden kann. Abbildung 3.3 bietet zur Übersicht über die im Folgenden beschriebenen Phasen ein Zustandsübergangsdiagramm für einen Knoten während des Schlüsselaustauschs.

Inquiry Phase

In der Inquiry Phase werden Vorbereitungen für das Schlüsselaustauschprotokoll getroffen. Dazu gehört das Synchronisieren mit dem entfernten Knoten und eine Rollenverteilung. Beides erfolgt durch den Austausch von *InquiryMsg*-Nachrichten. Das Format dieser Nachrichten sieht wie folgt aus:

```
typedef nx_struct InquiryMsg {
        nx_uint8_t type;
        nx_uint16_t nodeid;
        nx_uint16_t targetid;
} InquiryMsg;
```

Wird das `generateKey`-Command aufgerufen, wechselt der Knoten in die Inquiry-Phase. Solange er sich in dieser befindet, sendet er periodisch `InquiryMsg`-Nachrichten aus, bis er eine Antwort erhält und das Protokoll starten kann oder ein *Timeout*-Event signalisiert wird. Die periodischen `InquiryMsg`-Nachrichten enthalten die eigene `TOS_NODE_ID` im `nodeid`-Feld und die entfernte ID, die beim Aufruf des `generateKey`-Commands als Parameter übergeben wurde im `targetid`-Feld.

Empfängt ein Knoten in der Inquiry-Phase eine `InquiryMsg` vom entfernten Knoten, so wechselt er in die aktive Sampling-Phase und sendet eine `SampleMsg`-Nachricht. Er übernimmt somit die Rolle von Alice im Protokoll. Empfängt der andere Knoten die `SampleMsg`, so geht er in die passive Sampling-Phase (vgl. Abbildung 3.3) und übernimmt die Rolle von Bob. Im den weiteren Phasen werde ich den aktiven Knoten als Alice und den passiven als Bob bezeichnen.

Sampling Phase

In der Sampling-Phase tauschen Alice und Bob wechselseitig (Ping-Pong-Verfahren) `SampleMsg`-Nachrichten aus und speichern dabei den vom CC2420-Chip gemessenen RSSI-Werte. Wie in [33] beschrieben wird, eignet sich dieser Wert um den Zustand des Übertragungskanals zu repräsentieren. Wurden für einen Kanal `NUM_OF_SAMPLES` $\in \mathbb{N}$ Messungen durchgeführt, wird das arithmetische Mittel aller Samples für diesen Kanal berechnet und Alice fordert Bob auf, auf dem nächst höheren Kanal fortzufahren. Danach wechseln beide Parteien den Sende-/Empfangskanal und es werden erneut `NUM_OF_SAMPLES` Samples ausgetauscht. Um Nachrichten zu sparen, enthalten die Sample-Nachrichten alle nötigen Daten:

```
typedef nx_struct SampleMsg {
        nx_uint8_t type;
```

```
        nx_uint16_t nodeid;
        nx_uint16_t targetid;
        nx_uint8_t next_channel;
        nx_bool seqnr;
} SampleMsg;
```

Neben dem Header haben `SampleMsg`-Nachrichten das `next_channel`-Feld, welches den Übertragungskanal des nächsten Samples enthält. Dieses Feld wird von Alice dazu benutzt, Bob während der Sampling-Phase Anweisung zu geben, den Kanal zu wechseln. Ist die Qualität des Übertragungskanals zwischen Alice und Bob klein, kann es zum Verlust von Samples kommen. Um einen solchen Verlust zu erkennen, wurde das Feld `seqnr` eingeführt. Der Inhalt dieses Feldes ist induktiv definiert:

1. $\texttt{seqnr}_0 = \texttt{FALSE}$
2. $\texttt{seqnr}_{i+1} = \neg\texttt{seqnr}_i$ für $i \in \mathbb{N}$

Kommt es zu einem Verlust, wird dies beim Empfängerknoten durch ein Timeout-Ereignis erkannt. Dieser sendet dann das zuletzt gesendete Paket neu. So erhält der Knoten, dessen Paket verloren gegangen ist, eine Nachricht mit einer unerwarteten `seqnr` und sendet die verlorenen gegangene `SampleMsg` erneut.

Wurde auf allen verfügbaren Kanälen die durchschnittliche RSS gemessen, so wechselt Alice in die Key-Generation-Phase. Bei diesem Übergang sendet Alice eine `SettingsMsg`-Nachricht mit den Toleranzen für die Schlüsselgenerierung an Bob, woraufhin auch dieser in den nächsten Zustand wechselt.

Key Generation Phase

Nachdem Bob die `SettingsMsg` mit den Toleranzen empfangen hat, korrigiert dieser seinen Vektor $\texttt{avgrss}_\texttt{B}$ mit dem Mittelwerten der Messungen aus der Sampling-Phase und erhält so $\texttt{avgrss}'_\texttt{B}$. Sei `tol` der Vektor mit den Toleranzen für jeden Kanal (siehe hierzu Abschnitt 3.4.2) und $\texttt{RSS}_\texttt{min}$ die minimal wahrnehmbare Signalstärke, dann wird $\texttt{avgrss}'_\texttt{B}$ für jeden Kanal k nach folgender Funktion berechnet:

$$correct(\texttt{avgrss}_\texttt{B}[\texttt{k}], \texttt{tol}[\texttt{k}]) = \texttt{RSS}_\texttt{min} + \left\lfloor \frac{\texttt{avgrss}_\texttt{B}[\texttt{k}] - \texttt{RSS}_\texttt{min}}{2 \cdot \texttt{tol}[\texttt{k}]} \right\rfloor \cdot 2 \cdot \texttt{tol}[\texttt{k}] \qquad (3.2)$$

Der korrigierte Vektor $\texttt{avgrss}'_\texttt{B}$ dient als Geheimnis von Bob und wird nun kodiert, um den geheimen Schlüssel $\texttt{secret\_key}_\texttt{B}$ zu erhalten. Dazu wird die folgende Funktion `encode` verwendet, um von den korrigierten Messwerten auf ein 8-Bit Codewort zu kommen:

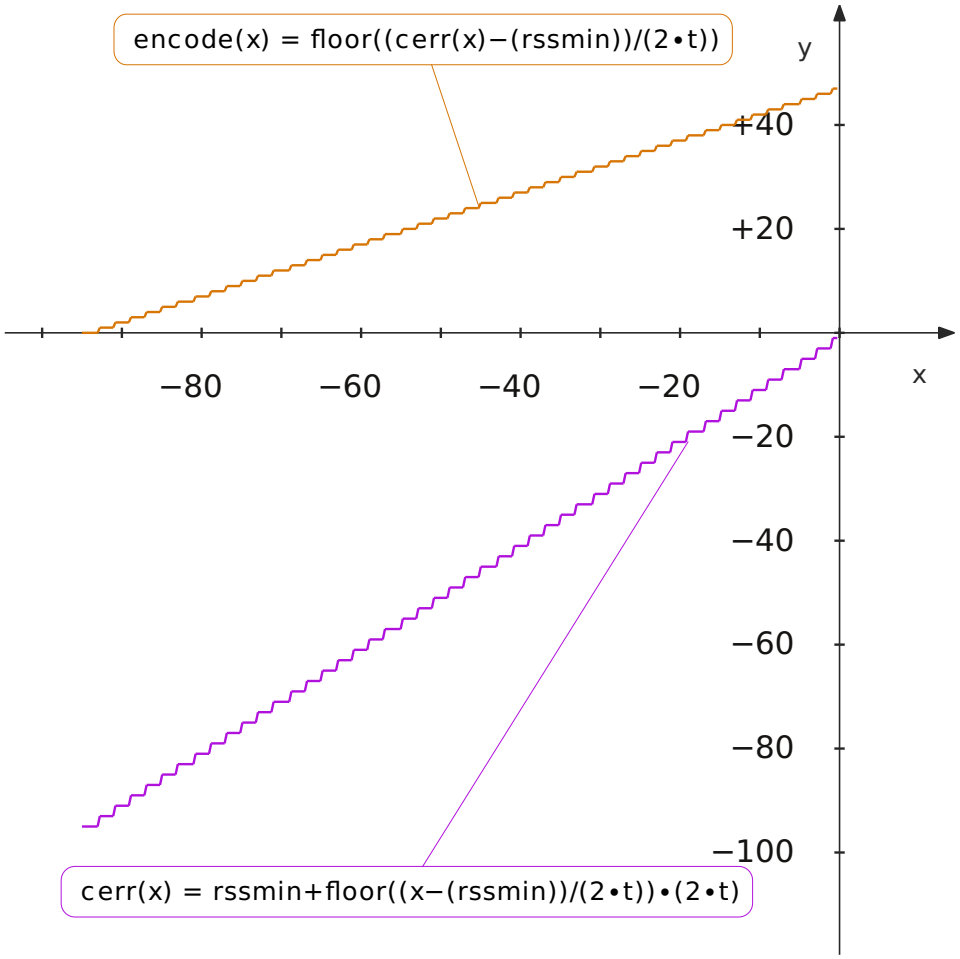


Abbildung 3.4: Zusammenhang zwischen der Fehlerkorrektur und der Kodierung. Die Funktion *cerr* ist die Fehlerkorrektur und die *encoding*-Funktion die Kodierung. Die Toleranz ist hier auf $t = 1$ und `rssmin` auf -95 gesetzt.

```
uint8_t encode (float plain, float tol) {
        return (uint8_t)floor((plain-RSSI_MIN)/(2*tol));
}
```

Der Zusammenhang zwischen der Fehlerkorrekturfunktion `correct` und der Kodierung `encode` ist in Abbildung 3.4 dargestellt.

Danach berechnet Bob einen Anpassungsvektor $\texttt{reconciliations} = \texttt{avgrss}'_B\text{-}\texttt{avgrss}_B$ und sendet diesen in einer `SettingsMsg` an Alice. Dieses Verfahren unterscheidet sich etwas von der Originalversion aus [37]. Dort wird dieser Anpassungsvektor nicht von Bob, sondern von Alice berechnet. Diese Änderung wurde hier vorgenommen um Nachrichten zu sparen. Von welcher Partei dieser Vektor nun berechnet wird macht von der mathematischen Sicht keinen Unterschied, da sich der Beweis aus Abschnitt

3.2.2 analog führen liesse wenn die Werte von Alice und Bob vertauscht werden.

Zum Austauschen der Toleranzen und des Anpassungsvektors zwischen Alice und Bob werden - wie schon erwähnt - `SettingsMsg`-Nachrichten verwendet. Diese haben folgendes Format:

```
typedef nx_struct SettingsMsg {
        nx_uint8_t type;
        nx_uint16_t nodeid;
        nx_uint16_t targetid;
        nx_float data[NUM_OF_CHANNELS];
} SettingsMsg;
```

TinyOS begrenzt die Größe des Payloads einer Active Message standardmäßig auf 28 Bytes. Auf MICAz-Knoten hat der Datentyp `float` für Fließkommazahlen eine Länge von 4 Bytes. So ergibt sich für `SettingsMsg`-Nachrichten eine Länge von 5 Bytes + $16 \cdot 4$ Bytes $= 69$ Bytes. Aus diesem Grund wird in der Header-Datei `KeyGen.h` die TinyOS-Variable `TOSH_DATA_LENGTH`, welche die maximale Payloadgröße reguliert auf die Größe einer `SettingsMsg` erhöht. Dies stellt kein Problem dar, solange die Größe einer Active Message (inkl. Metadaten der anderen Protokollschichten) nicht die CC2420-Grenze von 128 bit übersteigt [14]. Auf MICAz-Sensorknoten haben Active Messages z.B. einen Overhead von 10 Bytes und sind somit noch unter dem Limit [18].

Sobald Alice den Anpassungsvektor empfangen hat, passt sie ihre Messungen an, indem sie $\texttt{avgrss}_\texttt{A} + \texttt{reconciliations}$ berechnet und berechnet daraus mit dem gleichen Verfahren wie Bob das Codewort.

Security Indicator (SI): Wie in Abschnitt 3.3 beschrieben wurde, hängt die Sicherheit des Schlüssels von der verwendeten Toleranz ab. Um dem Anwendungsprogramm, welches den hier generierten Schlüssel verwenden soll, ein Feedback über die Sicherheit des Schlüssels zu geben, wird aus den hier verwendeten Toleranzen der *Security Indicator* $SI \in [0,1] \subset \mathbb{R}$ berechnet. Dieser gibt ein Verhältnis der Standardtoleranz zur tatsächlich verwendeten Toleranz an und wird wie folgt berechnet:

Sei $m \in \mathbb{N}$ die Anzahl der Kanäle. Seien weiter $t_{default}$ die anfangs verwendete Standardtoleranz und t_i die tatsächlich verwendete Toleranz des Kanals $i \in \{1, \ldots, m\}$. Dann gilt

$$SI = \frac{1}{m} \cdot \left(\sum_{i=1}^{m} \frac{t_{default}}{t_i} \right)$$

Wurden die Schlüssel mit der Standardtoleranz erfolgreich generiert, so ergibt sich ein SI von 1. Mussten aber alle Toleranzen im Toleranzwahlverfahren (siehe Abschnitt

3.4.2) verdoppelt werden ergibt sich

$$SI = \frac{1}{m} \cdot \left(\sum_{i=1}^{k} \frac{t_{default}}{2 \cdot t_{default}} \right) = \frac{1}{m} \cdot (m \cdot 0,5) = 0,5$$

Der *SI* verhält sich demnach antiproportional zur Summe der verwendeten Toleranzen. Dieser *SI* wird am Ende der `generateKey`-Operation mit dem Schlüssel an das `generationDone`-Event übergeben. Ein niedriger *SI*-Wert signalisiert dem Anwendungsprogramm eine geringe Sicherheit des Schlüssels. So können sicherheitskritische Anwendungen beispielsweise jeden Schlüsselaustausch mit einem $SI < 0,8$ als nicht erfolgreich betrachten.

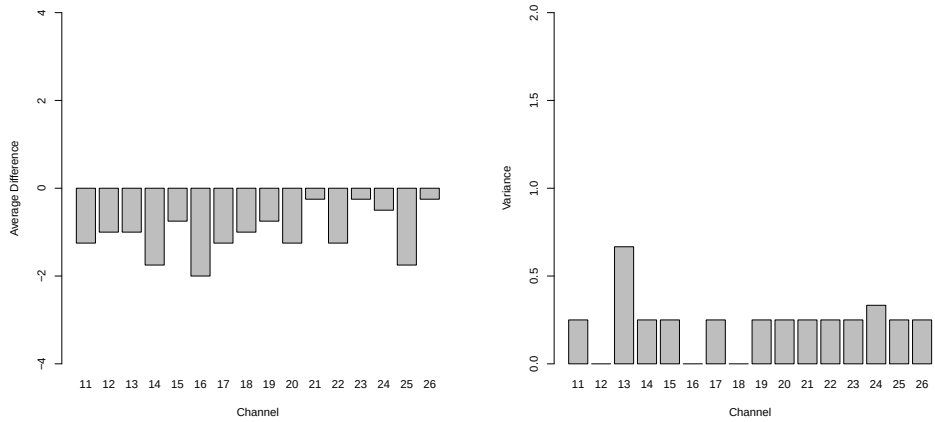
Nachdem in dieser Phase mit der beschriebenen Fehlerkorrektur und Kodierung Alice ebenfalls den Schlüssel und den *SI*-Wert berechnet hat, wechselt sie in die Acceptance Phase, welche sicherstellen soll, dass beide Parteien den gleichen Schlüssel generiert haben.

Acceptance Phase

Wenn Alice die Acceptance-Phase betritt, berechnet sie mit einer Hashfunktion den Hashwert ihres Codeworts. Dieses Schlüsselverfahren verwendet eine in nesC übersetzte Version des MD5-Algorithmus aus [26]. Mit diesem Algorithmus berechnet sie für ihr Codewort einen 128-Bit-Hashwert und sendet diesen in einer `DigestMsg`-Nachricht an Bob. Dieser geht beim Empfang dieses Hashwertes ebenfalls in die Acceptance-Phase über und berechnet ebenfalls eine MD5-Summe seines Codeworts. Den berechneten Hashwert vergleicht er dann mit dem empfangenen Hashwert und informiert Alice in einer `VerificationMsg` über das Ergebnis des Vergleichs. Die Strukturen der beiden Nachrichtentypen `DigestMsg` und `VerificationMsg` sind wie folgt definiert:

```
typedef nx_struct DigestMsg {
        nx_uint8_t type;
        nx_uint16_t nodeid;
        nx_uint16_t targetid;
        nx_uint8_t digest[16]; // MD5 Sum
} DigestMsg;

typedef nx_struct VerificationMsg {
        nx_uint8_t type;
        nx_uint16_t nodeid;
        nx_uint16_t targetid;
        nx_bool verified;
} VerificationMsg;
```



(a) Differenz zwischen Messungen von Alice und Bob

(b) Varianz der Messungen von Alice

Abbildung 3.5: Beispielmessung für einen Vergleich von Schwankungen und Varianz.

Das Feld `verified` einer `VerificationMsg` wird genau dann von Bob auf wahr gesetzt, wenn die beiden Hashwerte übereinstimmen. Stimmen diese nicht überein, so erhöht Alice anhand des in Abschnitt 3.4.2 beschriebenen Toleranzwahlverfahren die Toleranzen und wiederholt die Key Generation Phase.

Das Ende des Schlüsselaustauschs signalisiert Alice Bob, indem sie ihm eine `GenerationDoneMsg` sendet.

3.4.2 Toleranzwahlverfahren

Da die Wahl der Toleranzen direkten Einfluss auf den Erfolg und die Sicherheit des Protokolls hat, ist es wichtig optimale Werte zu finden. Die Implementierung arbeitet im ersten Durchlauf der Key Generation Phase mit den voreingestellten Toleranzen $t_{k,0}$ von Alice für jeden Kanal k. Wird in der Acceptance Phase aber festgestellt, dass die Schwankungen zu groß für die verwendeten Toleranzen waren (d.h. größer als t), dann müssen diese angepasst werden. Da eine möglichst hohe Sicherheit gewährleistet werden soll, sollen die Toleranzen dynamisch nur für die Kanäle erhöht werden, bei denen die zu hohen Schwankungen auftreten. Sie sollen dabei auch nur so weit erhöht werden, wie wirklich nötig ist.

Ein Ansatz ist, diese Kanäle über statistische Daten aus der Sampling Phase auszumachen. Würde man dazu einen geeigneten Weg finden, könnten die neuen Toleranzen ohne zusätzlichen Kommunikationsaufwand gewählt werden. Auch hat dies den

Vorteil, dass jeder Kanal völlig unabhängig betrachtet werden kann und somit für jeden Kanal die optimale Toleranz gefunden werden kann. Ein mögliches Kriterium ist die Varianz. Schwankungen in den Messwerten können beispielsweise daher rühren, dass manche Samples durch Interferenzen *beim Empfänger* gestört werden. Diese Interferenzen würden eine hohe Varianz hervorrufen, da die Messwerte der Samples stark vom Mittwelwert abweichen. Leider werden die Schwankungen aber zusätzlich noch durch eine Vielzahl anderer Faktoren wie Unterschiede in der Hardware beeinflusst. Wie in in Abbildung 3.5 dargestellt, lässt sich in dieser Beispielmessung zwischen den Unterschieden in den Messwerten von Alice und Bob und der Varianz der Samples von Alice keinen Zusammenhang erkennen. Abbildung 3.5a zeigt auf Kanal 16 eine relativ hohe Abweichung von 2 dBm, die dazugehörige Varianz der Messungen auf Kanal 16 in Abbildung 3.5b beträgt aber 0. Somit bleibt hier nur das Verwenden eines *empirischen Ansatzes* mit zusätzlichem Kommunikationsaufwand.

Um geeignete Toleranzen mit einer größtmöglichen Sicherheit zu finden, müssen folgende zwei Probleme gelöst werden:

1. Finden aller Kanäle k, bei denen die Schwankung größer als die verwendete Standardtoleranz $t_{k,0}$ war.
2. Wie groß sind die Schwankungen bzw. wie weit müssen die Toleranzen erhöht werden, damit der Schlüssel erfolgreich generiert wird.

Um diese beiden Problemstellungen empirisch zu lösen, ohne Eve Informationen über den Schlüssel preiszugeben, wurde hier das in Protokoll 2 in Pseudo-Code dargestellte Verfahren angewandt. Dieses Verfahren besteht aus zwei Phasen, die die genannten Probleme lösen.

In der ersten Phase wird die Toleranz auf allen Kanälen schrittweise erhöht und dabei überprüft, ob übereinstimmende Schlüssel generiert werden können. Ist dies der Fall, wurde die Toleranz t_{max} gefunden, die nötig ist, um die größte Schwankung in den Übertragungskanälen auszugleichen. Die Schrittweite (`TOLERANCE_INC_CONST` in Protokoll 2) kann dabei frei gewählt werden. Sie sollte aber nicht zu klein sein, da dies zu vielen Schritten und damit zu einem vergrößerten Kommunikationsaufwand führen kann. Wird die Schrittweite dagegen zu groß gewählt, geht zu viel Sicherheit verloren. Als Vorgabewert ist eine Schrittweite von $0,25$ eingestellt (siehe nächster Abschnitt).

In der zweiten Phase werden alle Kanäle gefunden, die mit mit der Standardtoleranz korrigiert werden können. Dazu werden für alle Kanäle k folgende Schritte wiederholt:

1. Setze Toleranz t_k für Kanal k auf die Standardtoleranz t_0. Für alle anderen Kanäle i wird t_i auf t_{max} gesetzt, falls Kanal i nicht markiert ist. Falls doch, setze t_i auf t_0.
2. Wiederhole Key Generation Phase und Acceptance Phase mit den neuen Toleranzen. Falls eine Übereinstimmung der Schlüssel erzielt wurde, markiere diesen Kanal.

```
// increase tolerances until agreement
tolerance = t_0
repeat
  tolerance += TOLERANCE_INC_CONST
  do tol[k] = tolerance for all k ∈ CHANNELS
until verifyKeyWithTol(tol)

// now find unnecessarily corrected channels
for all k ∈ CHANNELS do
  for all i ∈ CHANNELS do
    if i ≠ k then
      tol[i] = agree[i] ? tol[i] : t_0
    else
      tol[i] = tolerance
    end if
  end for

  agree[k] = verifyKeyWithTol(tol)
end for
```

Protokoll 2: Toleranzwahlverfahren. Die Funktion *verifyKeyWithTol*(tol) stellt hier vereinfachend die Ausführung der Key Generation Phase und der Acceptance Phase mit den als Parameter *tol* übergebenen Toleranzen dar. Ihr Rückgabewert ist ein boolescher Wert, der angibt, ob übereinstimmende Schlüssel generiert wurden. t_0 ist die Standardtoleranz.

Somit wurden nur die Toleranzen für diejenigen Kanäle auf t_{max} erhöht, deren Schwankung größer als t_0 war. Bei diesem Verfahren kommt es dennoch zu „Überkorrekturen“, d.h. es wird auf manchen Kanälen mehr korrigiert als nötig ist. Dieses Problem kommt daher, dass für jeden Kanal mit einer größeren Schwankung als t_0 diejenige Toleranz gewählt wird, die für den Kanal mit der höchsten Schwankung benötigt wird. Die damit zusammenhängende Einbuße an Sicherheit stellt eine Kompromisslösung zwischen Kommunikationsaufwand und Sicherheit dar. Eine Anpassung, die für jeden Kanal die optimale Toleranz findet, würde erheblich mehr Nachrichten benötigen. Der große Vorteil eines solchen dynamischen Toleranzwahlverfahrens liegt darin, dass der Schlüsselaustausch in jedem Fall erfolgreich beendet werden kann, sofern die Kommunikation zwischen Alice und Bob nicht unterbrochen wird.

3.4.3 Einstellungen/Optionale Features/Werkzeuge

Um das Protokoll an die individuellen Bedürfnisse anzupassen und um nähere Informationen zu bekommen, bietet die Implementierung der `KeyGenC`-Komponente

mehrere Konfigurations- und Erweiterungsmöglichkeiten. Diese werden mit Hilfe bedingter Übersetzung durch den Präprozessor von nesC gesteuert und können genau wie bei C-Programmen durch die Umgebungsvariable `CFLAGS` aktiviert werden. Folgende Optionen können aktiviert werden:

1. `DEBUG` - Optisches Feedback über die LEDs

   Dieses Flag schaltet die Verwendung der LEDs an. Bei MICAz-Knoten blinkt die gelbe LED beim Empfang einer Nachricht, die grüne LED beim erfolgreichen Senden und die rote LED bei jedem Kanalwechsel. Auf diese Weise bietet einem die Komponente hier die Möglichkeit den Verlauf des Protokolls grob zu verfolgen.

2. `SERIAL` - Protokollierung des Protokollablaufs über die serielle Schnittstelle des Programming Boards

   Ist dieses Flag gesetzt, werden alle Nachrichten, die der betreffende Knoten über Funk empfängt an den PC weitergeleitet. Weiter werden Daten wie erzeugter Schlüssel und SI-Wert an den PC gesendet. Mit Hilfe des beiliegenden Java-Tools *Reader* werden diese Daten von der seriellen Schnittstelle ausgelesen und lesbar ausgegeben. Dies erlaubt eine detailierte Beobachtung und Analyse des Protokolls. Dieses Tool verwendet TinyOS-Klassen und setzt eine vollständige und korrekte TinyOS-Installation voraus.

3. `CALIB` - Verwendung `Calib`-Komponente zur Kalibrierung des RSSI-Wertes

   Details sind Kapitel 4 zu entnehmen.

4. `DEFAULT_TOLERANCE` - Legt die Standardtoleranz t_0 fest

   Dieser Wert ist als Vorgabe auf $0,5$ gesetzt und kann mit dieser Option verändert werden.

5. `TOLERANCE_INCREASE` - Legt die Schrittweite des Toleranzwahlverfahren fest

   Als Vorgabe wird hier der Wert $0,25$ verwendet.

Die Vorgabewerte für diese Einstellungen können in der `KeyGen.h`-Datei verändert werden. Soll beispielsweise eine Anwendung für einen MICAz-Knoten kompiliert werden und dabei `DEBUG` und `SERIAL` gesetzt werden und zusätzlich eine Standardtoleranz von 1 verwendet werden, würde der `make`-Aufruf folgendermaßen aussehen:

```
make CFLAGS+="-DDEBUG -DSERIAL -DDEFAULT_TOLERANCE=1" micaz install,...
```

Sollen diese Optionen dauerhaft gesetzt werden, bietet es sich an folgende Zeile an den Anfang der Makefile für die TinyOS-Anwendung zu setzen:

```
CFLAGS += "-DDEBUG -DSERIAL -DDEFAULT_TOLERANCE=1"
```

4 Kalibrierung

Um den im vergangenen Kapitel beschriebenen Schlüsselaustausch erfolgreich und sicher einsetzen zu können, ist man auf genaue Messungen der empfangenen Signalstärke angewiesen. Durch die hohe Reziprozität des Übertragungskanals können mit Knoten, die ein einheitliches Sende- und Empfangsverhalten haben, Schlüssel mit niedrigen Toleranzen und somit hohen Sicherheiten generiert werden. Um die empfangene Signalstärke zu messen, wurde für das implementierte Protokoll der *Received Signal Strength Indicator* (RSSI) des auf den MICAz-Knoten verwendeten CC2420-Chips benutzt. Dieser bietet ein 8-Bit Register, welches den RSSI-Wert in Zweierkomplementdarstellung enthält. Dieser Wert stellt eine Schätzung der empfangenen Signalstärke dar und wird als Durchschnittsmessung über 8 Symbolperioden (128 µs) berechnet. Der Zusammenhang zwischen der empfangenen Signalstärke P und dem Registerwert `RSSI.RSSI_VALUE` ist durch

$$P = \texttt{RSSI\_VALUE} + \texttt{RSSI\_OFFSET}$$

gegeben, wobei $\texttt{RSSI\_OFFSET} \approx -45$ gilt.

Wie in [13] spezifiziert, hat der RSSI-Wert des CC2420-Transceiver eine Genauigkeit von $\pm 6\ dB$ [14]. Für unser Protokoll bedeutet das, dass sich wahrgenommene Signalstärken zwischen Knoten um bis zu 12 dB unterscheiden können. Im schlimmsten Fall würde das für den Schlüsselaustausch bedeuten, dass Toleranzen von mindestens 12 verwendet werden müssten um das Protokoll erfolgreich zu beenden. Wie Abschnitt 3.3 beschreibt, würde die Sicherheit des Schlüssels unter einem solchen Effekt enorm leiden. Die Tatsache, dass sich Knoten außer bei der Messung der RSS auch in der Leistung mit der gesendet wird unterscheiden, verstärkt diesen Effekt. Abbildung 4.1 zeigt das typische Verhalten des RSSI-Wertes für verschiedene Signalstärken. Die Schwankungen und Abweichungen sind deutlich an der Form des Graphen zu sehen.

Dies alles führt zu der Erkenntnis, dass eine Kalibrierung der Knoten für die Sicherheit dieses Protokolls essentiell ist. Eine automatisierte Kalibrierung von MICAz-Knoten wird in [22] beschrieben. Dazu werden die Knoten direkt an einen computergesteuerten Signalgenerator angeschlossen um die Sende- und Empfangscharakteristika eines Knoten genau zu messen. Auf der Basis dieser Messungen werden Kalibrierungstabellen erstellt, mit deren Hilfe die Abweichungen des RSSI-Wertes ausgeglichen werden. Auf diese Weise gelingt es, die Genauigkeit des RSSI auf $\pm 1,5\ dB$ zu erhöhen. Diese Kalibrierung ist aber durch den zusätzlichen Hardwareaufwand (Signal Generator, Verbindungskabel) und komplexe Berechnungen recht aufwändig. Die optimale Lösung wäre, die Knoten bei der Herstellung im Werk so genau

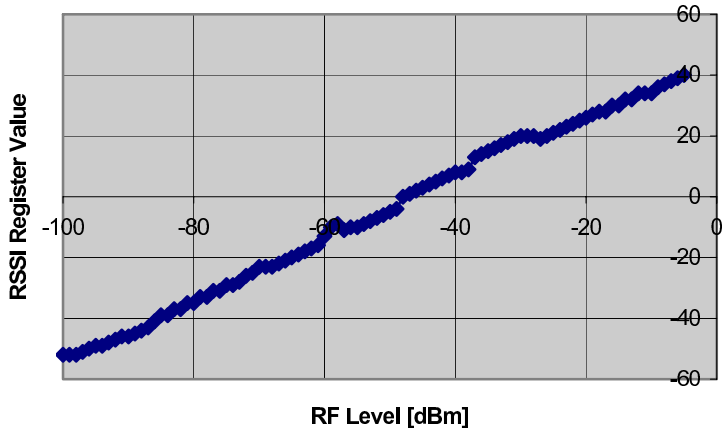
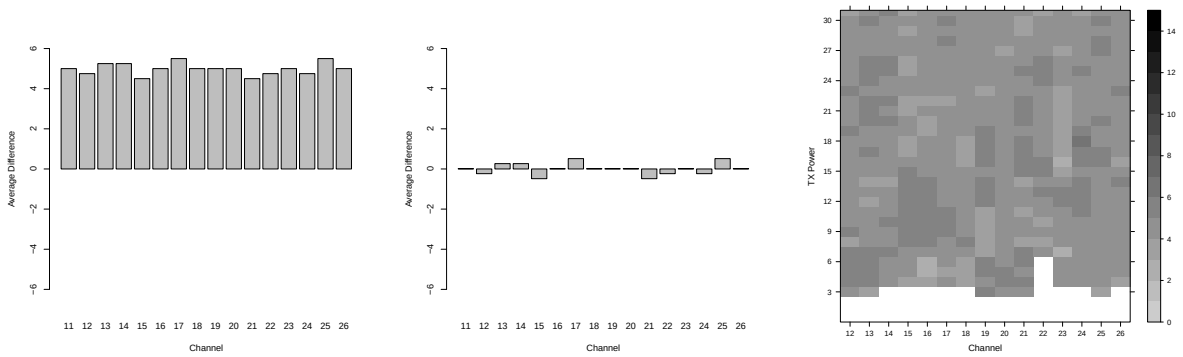


Abbildung 4.1: Typischer Verlauf des RSSI-Wertes bei verschiedenen empfangenen Signalstärken [14]

wie möglich zu kalibrieren. Dann würden diese Probleme hier keine Rolle spielen. Da dies aber nicht der Fall ist und solche Schwankungen auch von nachträglichen Defekten an der Hardware herrühren können, muss eine Kalibrierung nachträglich möglich sein.

Für das Protokoll selbst können die Anforderungen an die Kalibrierung etwas eingeschränkt werden. Die einzige Forderung an die verwendeten Knoten ist, dass sie Kanaleigenschaften einheitlich wahrnehmen sollen um so das gemeinsame Geheimnis besser extrahieren zu können. Eine Übereinstimmung des RSSI-Wertes mit der tatsächlich wahrgenommenen Leistung ist dagegen nicht zwingend nötig. Die hier geforderte Kalibrierung muss daher lediglich die Funktionen f_A und f_B, die die in Abbildung 4.1 dargestellte Kurve für zwei Knoten A und B repräsentieren angleichen. Genau wie bei dem Schlüsselaustausch ist hier ein weiteres Ziel die einfache Verwendbarkeit. Es soll kein zusätzliches Equipment erforderlich sein und der Benutzer soll die Kalibrierung einfach und schnell durchführen können.

Um diese Ziele zu erreichen, wird hier eine *grobe* Kalibrierung implementiert. Sie versucht den durchschnittlichen Unterschied in den Kurven der Knoten herauszufinden und auszugleichen. Das Prinzip der Kalibrierung ist in Abbildung 4.2 dargestellt. Abbildung 4.2a zeigt dabei die Differenzen in RSS-Messungen des Kanals zwischen zwei Knoten für alle Kanäle. Die durchschnittliche Abweichung beträgt in diesem Beispiel $4,98$. Subtrahiert man diese von allen Kanälen, erhält man die in Abbildung 4.2b dargestellten Werte. Vor dieser Korrektur betrug die größte Abweichung $5,5$ dB. Das subtrahieren des Mittelwerts reduziert diese maximale Abweichung auf $0,52$, was den Schlüsselaustausch erheblich verbessern würde. Dieser Mittelwert scheint also ein geeigneter Wert für die Kalibrierung zu sein. Wie man in Abbildung 4.1 erkennen kann, unterscheidet sich die Höhe der Schwankung für verschiedene RSS-Werte. Experimente haben jedoch gezeigt, dass dieser Effekte relativ gering ist und somit die Schwankungen über verschiedene Empfangsstärken in den meisten Fällen ausreichend stabil sind (vgl. Abbildung 4.2c) um sie mit der mittleren Schwankung



(a) Unterschiede in den RSS-Messungen zweier Knoten A und B

(b) Unterschiede abzüglich der durchschnittlichen Abweichung

(c) Unterschiede in den Messungen bei verschiedenen Sendestärken

Abbildung 4.2: Prinzip der Korrektur: Subtrahiere die durchschnittliche Abweichung von allen Kanälen (hier: 4, 98) von der gemessenen RSS. Die hier dargestellten Werte beziehen sich auf die Sicht von A.

zu korrigieren.

Die im Folgenden vorgestellte Kalibrierung versucht diese mittlere Schwankung zwischen zwei Knoten herauszufinden. Diese wird dann dauerhaft auf dem Knoten gespeichert und kann später zum Korrigieren der Messungen während des Schlüsselaustauschprotokolls verwendet werden.

4.1 Implementierung

Die Kalibrierung besteht aus zwei Komponenten. Zum einen der `CalibrationApp`-Anwendung, welche die oben beschriebene durchschnittliche Abweichung zwischen zwei Knoten herausfinden soll und auf dem Flash-Speicher des Sensorknotens speichert. Die zweite Komponente ist die `CalibC`-Komponente, welche das Interface `Calib` implementiert. Dieses Interface bietet Funktionen zum Ändern und Verwenden der Kalibrierungsdaten auf dem Flash-Speicher. Das Interface und dessen Implementierung werden später in Anwendungen (wie z.B. dem Schlüsselaustauschprotokoll) verwendet, um die Messungen anzugleichen.

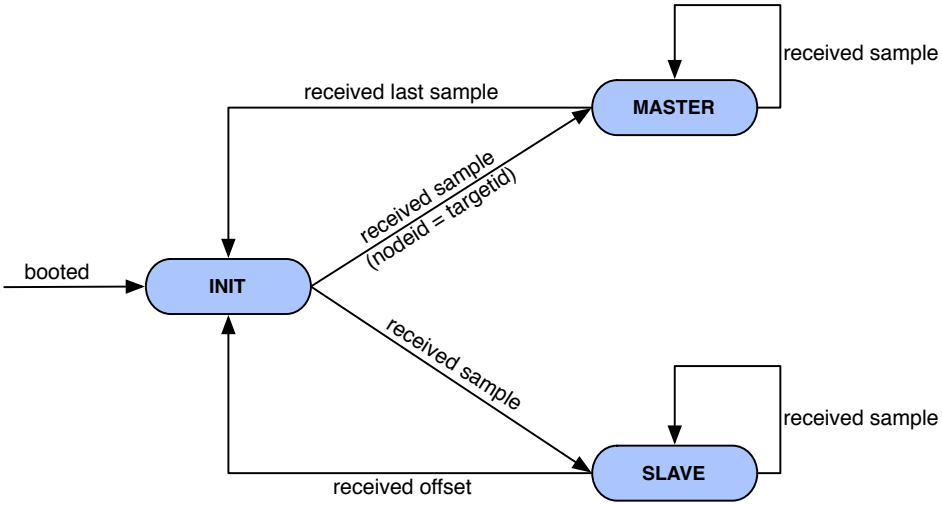


Abbildung 4.3: Zustandsübergangsdiagramm der `CalibrationApp` Komponente

4.1.1 CalibrationApp

Ähnlich wie beim vorgestellten Schlüsselaustausch werden auch bei dieser Kalibrierung Nachrichten ausgetauscht und dabei die RSS gemessen. Abbildung 4.3 zeigt dazu das Zustandsübergangsdiagramm der `CalibrationApp`-Anwendung. Während den Messungen wird zwischen einem *Master*-Knoten und einem *Slave*-Knoten unterschieden. Ersterer steuert die Messungen und berechnen am Ende der Messungen die Kalibrierungsdaten. Slave-Knoten dagegen lesen beim Empfang einer Nachricht lediglich den RSSI-Wert und schicken diesen zum Master-Knoten zurück. Alle in dieser Anwendung ausgetauschte Nachrichten haben folgende Struktur:

```
typedef nx_struct CalibrationMsg {
        nx_uint16_t nodeid;
        nx_uint16_t targetid;
        nx_uint8_t seqnr;
        nx_uint8_t next_channel;
        nx_float data;
} CalibrationMsg;
```

Da hier nur eine Art von Nachrichten ausgetauscht wird, wird kein Typ-Feld benötigt. Die Felder `nodeid` und `targetid` dienen wie bisher der Addressierung der Nachrichten. Durch die Sequenznummer `seqnr` wird sichergestellt, dass alle Samples erfolgreich ausgetauscht werden. Weiter gibt das `next_channel`-Feld dem Slave-Knoten an, auf welchem Kanal er senden und empfangen soll und das `data`-Feld enthält den RSSI-Wert eines Sample-Austauschs vom Slaveknoten.

Nachdem der Sensorknoten gestartet wurde und auf den Startkanal 11 gewechselt hat, sendet er eine `CalibrationMsg`, wobei er `nodeid` und `targetid` auf seine `TOS_NODE_ID` setzt. Empfängt ein anderer Knoten im `INIT`-Zustand diese Nachricht, wechselt dieser in den `MASTER`-Zustand und startet eine Kalibrierung indem er seinerseits mit einer `CalibrationMsg` antwortet. Nun tauschen die beiden Knoten auf jedem verfügbaren Kanal `SAMPLES_PER_CHANNEL` $\in \mathbb{N}$ Samples aus. Der Master-Knoten steuert dabei den Slave, indem er das `next_channel`-Feld der Samples und ihre Sequenznummer setzt. Der Slave sendet jede eingegangene Nachricht zurück und setzt dabei das `data`-Feld immer auf die RSSI der empfangenen Nachricht. Der Master speichert seinerseits bei jeder empfangenen Nachricht die Differenz d_i ($i = 0, \ldots,$ `SAMPLES_PER_CHANNEL`) zwischen dem im `data`-Feld enthaltene RSSI-Wert des Slave-Knoten und dem RSSI-Wert der Nachricht. Jedesmal wenn `SAMPLES_PER_CHANNEL` Samples auf einem Kanal k ausgetauscht wurden, berechnet der Master-Knoten den Mittelwert μ_k aller d_i ($i = 0, \ldots,$ `SAMPLES_PER_CHANNEL`) und wechselt auf den nächsten Kanal.

Nachdem dies für alle Kanäle k ($k = 1, \ldots,$ `NUM_OF_CHANNELS`) wiederholt wurde, wird von den Mittelwerten μ_k ebenfalls das arithmetische Mittel μ errechnet. Dieses μ stellt den gesuchten Mittelwert dar. Der Master-Knoten sendet zum Abschluss der Kalibrierung eine Nachricht, welche diesen Mittelwert enthält. Da die Schwankungen oftmals von defekten Transmittern herrühren, die eine geringere Signalstärke als intakte Transmitter abstrahlen, wird das Offset auf dem intakten Knoten gespeichert, damit dieser bei späteren Messungen die RSSI-Werte des zu schwachen Signals korrigieren kann. Welcher der intakte Knoten ist, hängt von dem Vorzeichen von μ ab. Ist das Vorzeichen negativ, waren die Differenzen im Durchschnitt negativ. Das bedeutet, dass der Master im Durchschnitt ein stärkeres Signal gemessen hat. Somit speichert der Slave-Knoten μ und der Master-Knoten 0 als Kalibrierungswert.

Features

Um den Vorgang des Kalibrierens überwachen zu können, kann das beiliegende Java-Programm *Reader* verwendet werden. Dazu muss beim Kompilieren das Flag `SERIALDBG` gesetzt werden, welches die Weiterleitung aller empfangenen oder gesendeten Nachrichten an den PC aktiviert. Um die Genauigkeit zu erhöhen sollte die Kalibrierung unter Beobachtung mehrmals durchgeführt werden und bei einem häufig auftauchenden Kalibrierungswert belassen werden.

Da Sensorknoten in unterschiedlichen WSNs eingesetzt werden und Knoten-IDs geändert werden, muss es möglich sein die Kalibrierungsdaten zu löschen. Dies geschieht durch das Präprozessor-Flag `ERASE`. Ist es gesetzt, werden die Daten nach dem Booten vom Flash-Speicher des Knoten gelöscht. Zum erneuten Kalibrieren darf das Flag nicht mehr gesetzt sein.

```
typedef struct calibration_config_t {
        float mapping[MAX_NODEID];
} calibration_config_t;

interface Calib {
        command error_t init ();
        event void initDone (error_t error);
        command float mapRssi (float rssi, uint16_t nodeid);
        command void erase ();
        command void setCalibration (calibration_config_t* calib);
        command void getCalibration (void* target);
}
```

Listing 4.1: Calib Interface

4.1.2 Calib/CalibC

`CalibC` implementiert das in Listing 4.1 dargestellte Interface `Calib`. Dieses stellt zum einen eine Split-Phase-Operation `init` bereit, welche die von der `CalibrationApp`-Anwendung gespeicherte Kalibrierungs-Offsets ausliest. Diese sind auf dem Flash-Speicher in einer `calibration_config_t`-Struktur gespeichert. Die Commands `erase`, `setCalibration` und `getCalibration` ermöglichen einen Lese- und Schreibzugriff auf gespeicherte Kalibrierungs-Offsets. Die Konstante `MAX_NODEID` gibt die Anzahl der Knoten, mit denen kalibriert werden kann an. Als Standardwert ist diese auf 25 gesetzt. Das bedeutet, dass die Kalibrierung für alle Sensorknoten mit einer `TOS_NODE_ID` $\in \{1,\ldots,25\}$ funktionieren würde. Für größere WSNs müsste `MAX_NODEID` entsprechend verändert werden. Dies kann über den Präprozessor des nesC-Compiler gemacht werden (vgl. 3.4.3).

Wesentlich wichtiger für Anwendungen ist das Command `mapRssi`. Die zwei Parameter `rssi` und `nodeid` werden hier auf eine Gleitkommazahl abgebildet, indem `rssi` und das für den Knoten `nodeid` gespeicherte Offset addiert werden. Auf diese Weise werden RSSI-Werte „korrigiert“.

Für das Beispiel aus Abbildung 4.2 würde die Kalibrierung und der kalibrierte Schlüsselaustausch folgendermaßen Ablaufen:

1. Die `CalibrationApp`-Anwendung würde auf Knoten A den Wert $4,98$ als Offset für Knoten B speichern.
2. Die `KeyGenC`-Komponente würde beim Schlüsselaustausch während der Key Generation Phase nicht direkt die $\texttt{avgrss}_\texttt{A}$-Werte benutzen, sondern $\texttt{mapRssi}(\texttt{avgrss}_\texttt{A},\texttt{B})$. Auf diese Weise werden die Schwankungen auf die in Abbildung 4.2b dargestellten Werte reduziert. Da die maximale Schwankung dann nur noch $0,52$ beträgt, wäre der Schlüsselaustausch nach der ersten Toleranzerhöhung erfolgreich.

5 Evaluierung

In diesem Kapitel werden die erreichten Ziele quantifiziert. Im ersten Teil dieses Kapitels wird die `KeyGenC`-Komponente auf ihre Effizienz und die Sicherheit des generierten Schlüssels untersucht. Der zweite Teil beschäftigt sich mit der Kalibrierung. Anhand des Security Indicators von generierten Schlüsseln wird die Verbesserung durch eine Kalibrierung betrachtet.

5.1 Schlüsselgenerierung

Eines der Hauptziele des Protokolls ist es, einen Schlüsselaustausch bereit zu stellen, der an die begrenzten Ressourcen wie Speicher und Rechenleistung von WSNs angepasst ist. Der größte Rechenaufwand liegt hier bei der eingesetzten Fließkommaarithmetik, da diese nicht direkt durch die Hardware unterstützt wird und somit auf Software-Ebene durchgeführt werden muss. Um nun den Rechenaufwand zu beurteilen, wird im folgenden die Anzahl der durchgeführten Fließkommaoperationen berechnet:

Sei m die Anzahl der Kanäle und n die Anzahl der pro Kanal ausgetauschten Samples.

1. Sampling Phase:
    - Berechnung des arithmetischen Mittels aller Samples pro Kanal: Es müssen $m \cdot n$ Additionen und m Divisionen ausgeführt werden.
2. Key Generation Phase:
    - Fehlerkorrektur: Für jeden korrigierten Wert müssen 6 arithmetische Operationen durchgeführt werden (siehe *correct* in Abschnitt 3.4.1). Bei m Kanälen entspricht dies $m \cdot 6$ Operationen.
    - Kodierung: Zum Abbilden der korrigierten Werte auf den geheimen Schlüssel werden $m \cdot 4$ Operationen ausgeführt.
    - Berechnen des SI: Für jeden Kanal wird das Verhältnis von Standardtoleranz zur eingesetzten Toleranz berechnet und davon das arithmetische Mittel berechnet. Es werden demnach $m + 1$ Divisionen und m Additionen durchgeführt.
3. Acceptance Phase:

- Toleranzwahlverfahren: Die anfängliche Toleranzerhöhung auf allen Kanälen benötigt $i \cdot m$ Additionen, wobei

$$i = \left\lceil \frac{\text{max. Schwankung} - \texttt{t}_0}{\texttt{TOLERANCE\_INC\_CONST}} \right\rceil$$

  gilt.

Unter der Annahme, dass die Schwankungen kleiner oder gleich der Standardtoleranz sind (d.h. keine Toleranzwahl nötig) werden auf dem MICAz-Knoten mit $m = 16$ und $n = 20$

$$\underbrace{(m \cdot (n+1))}_{\text{Sampling Phase}} + \underbrace{(m \cdot (6+4+2)+1)}_{\text{Key Generation Phase}} = 529$$

Fließkommaoperationen durchgeführt.

Das Verwenden der `KeyGenC`-Komponente in einer TinyOS-Anwendung verbraucht auf MICAz-Knoten ca. 24,25 KiB des 128 KiB großen Programm-Speichers mehr. Es sind demnach noch ca. 81% des Speichers für die Anwendung verfügbar. Von den 4 KiB RAM, die auf MICAz-Knoten verfügbar sind, werden durch das Protokoll zusätzliche 795 Bytes belegt. Will man im Vergleich dazu das von TinyECC bereitgestellte ECDH-Verfahren zum Schlüsselaustausch verwenden, benötigt man ca. 1,7 KiB RAM [21]. Bei diesen Angaben ist aber noch nicht der dynamische Stack-Speicher mit eingerechnet, auf welchem lokale Variablen während der Laufzeit abgelegt werden. Beachtet man diesen, ist wegen des hohen RAM-Verbrauchs bei TinyECC kaum Speicher für größere Anwendungen verfügbar. Die `KeyGenC`-Anwendung kommt während der Ausführung auf eine maximale Stackgröße von ca. 54 Bytes. Somit sind noch mind. 79,27% der 4 KiB RAM während der Ausführung für andere TinyOS-Komponenten verfügbar.

5.1.1 Kommunikationsaufwand

Ein weiterer nicht vernachlässigbarer Aufwand für das Protokoll ist der Kommunikationsaufwand. Das Senden und Empfangen von Nachrichten kostet Energie und Energie ist wichtig in WSNs. Um die Anzahl der Nachrichten und die Menge an übertragenen Daten herauszufinden, werden die Phasen des Schlüsselaustauschs zuerst einzeln betrachtet. Die im Folgenden angegebenen Größen beziehen sich nur auf den Payload der Nachrichten. Zur Vereinfachung wird hier von einem optimalen Verlaufen des Protokolls (sofortiger Start, keine Toleranzwahl) ausgegangen. Das bedeuted, dass die berechneten Zahlen *Untergrenzen* darstellen.

1. **Inquiry Phase:** Hier wird jeweils eine `InquiryMsg`-Nachricht (5 Bytes) versendet und empfangen.
2. **Sampling Phase:** In dieser Phase werden die meisten Nachrichten ausgetauscht. Auf allen m Kanälen werden `NUM_OF_SAMPLES` `SampleMsg`-Nachrichten (7 Bytes)

ausgetauscht. Somit werden insgesamt $2 \cdot m \cdot$ `NUM_OF_SAMPLES` Nachrichten ausgetauscht.

3. **Key Generation Phase:** Diese Phase wird mit einer `SettingsMsg` (69 Bytes) von Alice eingeleitet und durch eine von Bob beendet. Somit also 2 Nachrichten.

4. **Acceptance Phase:** In dieser Phase wird eine `DigestMsg` (6 Bytes) von Alice an Bob gesendet, woraufhin dieser mit einer `VerificationMsg` (6 Bytes) antwortet. Beide Knoten schließen dann mit einer `GenerationDoneMsg` (4 Bytes) den Schlüsselaustausch ab.

Insgesamt werden also mindestens $8 + 2 \cdot m \cdot$ `NUM_OF_SAMPLES` Nachrichten ausgetauscht. Mit $m = 16$ und `NUM_OF_SAMPLES` $= 20$ wären es demnach 648 Nachrichten und einer Brutto-Datenmenge von 2408 Bytes. Da aber außer den Nutzdaten noch der in Abschnitt 3.4.1 erwähnte Overhead von 10 Bytes pro Nachricht anfallen, werden insgesamt $2408 + 648 \cdot 10 = 8888$ Bytes übertragen. Bei einer Übertragungsgeschwindigkeit von 250 kbit/s entspricht dies einer reinen Übertragungszeit von 284, 416 ms.

Dieser optimale Verlauf der Protokolls stellt aber eher die Ausnahme dar. Je nach Qualität des Funkkanals kommt es häufiger zum Verlust von Nachrichten, welche dann erneut übertragen werden müssen. Auch der Einsatz des Toleranzwahlverfahrens erzeugt zusätzlichen Kommunikationsaufwand. Dieser hängt dann - wie schon erwähnt - von der maximalen Schwankung ab. All diese Faktoren spielen für den Kommunikationsaufwand eine Rolle und daher ist es recht schwer, diesen genau vorherzusagen.

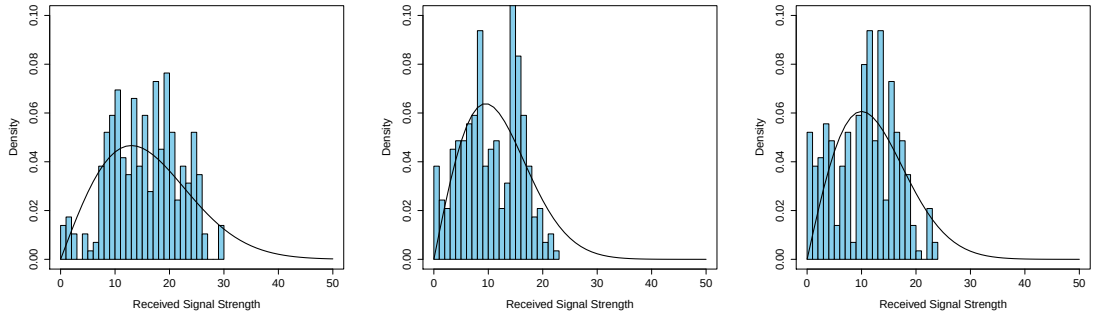
5.1.2 Sicherheit

In diesem Abschnitt wird die Sicherheit des Schlüsselaustauschprotokolls, wie sie in Abschnitt 3.3 beschrieben ist, analysiert. Dazu wird zuerst versucht diese zu quantifizieren. Gemessen werden kann die Sicherheit eines Schlüssels am Aufwand, der betrieben werden muss, um diesen vorherzusagen. Wie in 3.3 gezeigt wurde, hängt dieser Aufwand direkt von den verwendeten Toleranzen und der Verteilung der gemessenen Werte ab.

Ein Maß, welches diese Anforderungen erfüllt, ist der mittlere Informationsgehalt der Schlüssel, ihre Entropie. Die klassische Entropie nach Shannon $H(X)$ ist ein Maß für die Unbestimmtheit einer Zufallsvariable X und ist wie folgt definiert [31]:

$$H(X) = -\sum_{i=1}^{n} p(x_i) \cdot log p(x_i)$$

Der Wert n ist die Anzahl aller möglichen Werte von X und x_i ist der i-te Wert davon. Die Einheit der Entropie hängt hierbei von der Basis des verwendeten Logarithmus ab. Ist diese 2, so ist die Einheit bit. Im Folgenden ist die Basis immer 2.



(a) RSS-Verteilung auf Kanal 12 ($\sigma = 13$)

(b) RSS-Verteilung auf Kanal 20 ($\sigma = 10$)

(c) RSS-Verteilung auf Kanal 25 ($\sigma = 9,5$)

Abbildung 5.1: Beispiel für die Ähnlichkeit der RSS-Verteilung auf verschiedenen Übertragungskanälen mit der Rayleigh-Verteilung mit verschiedenen Parametern σ.

Um den Zusammenhang der Entropie mit den hier generierten Schlüsseln herzustellen, wird noch einmal das in 3.3 beschriebene Szenario betrachtet. Unter der Annahme, dass die RSSI-Werte gleichverteilt sind und die Toleranz $0,5$ beträgt, müsste Eve $7,94 \cdot 10^{26}$ Möglichkeiten durchprobieren, da jede mit der gleichen Wahrscheinlichkeit auftreten könnte. Die Entropie wäre hier maximal, da jedes Bit unbestimmt wäre, d.h. die Wahrscheinlichkeit für eine 0 die gleiche ist wie für eine 1: $p(0) = p(1) = 0,5$. Sei nun X die gleichverteilte Zufallsvariable über dem Alphabet der möglichen Schlüssel für eine Toleranz von $t = 0,5$. Dann ist die Entropie von X

$$\begin{aligned} H(X) &= -\sum_{i=1}^{7,94 \cdot 10^{26}} (7,94 \cdot 10^{26})^{-1} \cdot log((7,94 \cdot 10^{26})^{-1}) \\ &= -((7,94 \cdot 10^{26}) \cdot (7,94 \cdot 10^{26})^{-1} \cdot log((7,94 \cdot 10^{26})^{-1})) \\ &= -log((7,94 \cdot 10^{26})^{-1}) \\ &= log(7,94 \cdot 10^{26}) = 89,36\,\text{bit} \end{aligned} \tag{5.1}$$

Die Entropie ist im Kontext dieser Arbeit also die Anzahl an Bit, die nötig ist, um alle Möglichkeiten darzustellen, die ein Angreifer ohne weitere Informationen bei einer Brute-Force-Suche durchprobieren müsste (vgl. Gleichung 5.1). Da ich hier vom Optimalfall - einer Gleichverteilung der Schlüssel - ausgehe, bildet die Entropie von $89,36$ bit aus Gleichung 5.1 die Obergrenze. Da die RSSI-Werte und somit die Schlüssel aber wie oben erwähnt nicht gleichverteilt sind, sondern bestimmte Werte mit einer höheren Wahrscheinlichkeit auftreten, wird diese Entropie in der Realität niemals erreicht. Statt einer Gleichverteilung folgen die RSSI-Werte mehr einer Rayleigh-Verteilung, wie für ein Beispielexperiment mit 9 verschiedene Sensorknoten auf 9 verschiedenen Positionen in Abbildung 5.1 dargestellt ist [37].

$SI \in$	$[0, 0.5)$	$[0.5, 1]$	$[0, 1]$
n	27	173	200
Entropie	23,74	82,12	79,78

Tabelle 5.1: Empirische Shannon-Entropie [bit] von Schlüsseln, die auf zufälligen Positionen erzeugt wurden und deren SI sich im angegebenen Bereich befand. n gibt dabei die Anzahl der Schlüssel an. Die Standardtoleranz war bei allen Schlüsseln auf $0,5$ gesetzt.

Entropieschätzung

Mit der Entropie scheint also ein geeignetes Maß für Sicherheit gefunden zu sein. Der Nachteil der Entropie ist aber, dass alle Wahrscheinlichkeiten bekannt sein müssen, um diese zu berechnen. In diesem Fall können diese aber lediglich geschätzt werden, da ihre genaue Verteilung nicht bekannt ist. Zu diesem Zweck wurden in einem Experiment 200 Schlüssel von zufälligen Positionen generiert. Diese Schlüssel bestehen jeweils aus 16 8-Bit-Folgen (siehe hierzu 3.4.1), die jeweils das Geheimnis auf einem Kanal repräsentieren. Dieses Geheimnis ist die für uns interessante Zufallsvariable X über dem Alphabet aller auftretenden 8-Bit-Folgen. Es wird also immer eine 8-Bit-Folge zu einem Symbol zusammengefasst. Die Wahrscheinlichkeit für das Auftreten eines Symbols x aus dem Alphabet von X wird nun geschätzt, indem das Verhältnis der Anzahl der Vorkommen von x zu der Gesamtanzahl von Symbolen in den 200 Schlüsseln berechnet wird. Mit diesen geschätzten Wahrscheinlichkeiten wurden die in Tabelle 5.1 gezeigten empirischen Entropien für Schlüssel mit verschiedenen SIs berechnet.

Die in Tabelle 5.1 aufgelisteten Entropien wurden ohne Berücksichtigung von Abhängigkeiten zwischen den Symbolen, d.h. ohne Berücksichtung von Abhängigkeiten in den Messungen berechnet. Tatsächlich jedoch hängen die Symbole eines Schlüssel in gewisser Weise voneinander ab. Betrachtet man sich die Signal-Maps in Abbildung 3.1 auf Seite 10 noch einmal, so kann man erkennen, dass die gemessenen Signalstärken auf benachbarten Kanälen sehr nah beieinander liegen. So ist die Wahrscheinlichkeit hoch, dass die RSS auf Kanal 20 beispielsweise, nur wenige dB unter oder über der auf Kanal 19 und 21 liegt. Dies zeigt, dass die Werte voneinander abhängen. Messungen haben ergeben, dass der Unterschied zwischen der größten und der kleinsten gemessenen RSS selten größer als 25 dB ist. Diese Tatsache muss bei der Analyse der Sicherheit beachtet werden, da dadurch Sicherheit verloren geht.

Um die tatsächliche Entropie unter Beachtung aller Abhängigkeiten zu schätzen, werden hier Verfahren aus der algorithmischen Informationstheorie verwendet. Der Unterschied zur klassischen Informationstheorie ist, dass hier der Informationsgehalt nicht anhand der Entropie gemessen wird, sondern anhand der *Kolmogorov Komplexität*. Die Kolmogorov Komplexität ist einfach ausgedrückt ein Maß für die Komprimierbarkeit einer Zeichenfolge [20]. Wie mehrere Quellen (z.B. [30, 10]) zeigen, sind

$SI \in$	$[0, 0.5)$	$[0.5, 1]$	$[0, 1]$
n	27	173	200
T-Entropie	$15,06$	$55,74$	$51,67$

Tabelle 5.2: T-Entropie [bit] von Schlüsseln deren SI im angegebenen Intervall lag (Standardtoleranz $t_0 = 0,5$)

die Begriffe Kolmogorov Komplexität und Entropie eng miteinander verwandt. Wie oben erwähnt wurde, ist die Entropie im Kontext dieser Arbeit die kleinste Anzahl an Bit, die nötig ist um alle möglichen Schlüssel darzustellen. Dies zeigt den Zusammenhang mit der Kolmogorov Komplexität, da diese ebenfalls versucht die kleinste Darstellungsform für einen Schlüssel zu finden. Ein großer Vorteil dieser Komplexität für diese Arbeit ist, dass die tatsächliche Entropie der Schlüssel geschätzt werden kann, ohne die genaue Verteilung der RSS-Werte zu kennen.

Um die Entropie zu schätzen, führen Titchener et al. [35] den Begriff der *T-Complexity* einer Zeichenfolge ein. Sie stellt eine gewichtete Anzahl von Herstellungsschritten dar, die nötig ist um eine Zeichenkette von ihrem Alphabet zu konstruieren. Wie Speidel et al. [32] weiter zeigen, konvergiert die T-Complexity am schnellsten zum wahren Wert der Shannon Entropie. Um nun die Entropie mit Hilfe der T-Complexity zu schätzen, bietet dieselbe Gruppe das Tool `tcalc` an. Es hat als Eingabe eine Byte-Folge und berechnet deren T-Complexity. Mit diesem Tool wird im Folgenden die T-Complexity berechnet.

Da `tcalc` auf Byte-Folgen arbeitet, müssen die Schlüssel wie bei der vorherigen Analyse in ihre 8-Bit-Folgen zerlegt werden. Indem jede dieser 8-Bit-Folgen als eindeutiges Symbol kodiert wird, erhält man einen sogenannten *T-String*. Dieser T-String dient als Eingabe für `tcalc`. Tabelle 5.2 enthält die mit `tcalc` geschätzten *T-Entropien* der 200 zufälligen Schlüssel. Ein Angreifer, der die genaue Verteilung der Schlüssel kennt, müsste also bei Verwendung von Schlüsseln mit einem SI $> 0,5$ und einer Brute-Force-Attacke ca. 6 Mio. Schlüssel durchprobieren, um mit einer hohen Wahrscheinlichkeit den richtigen zu finden. Aufgrund der Unvorhersagbarkeit des Übertragungskanals zwischen zwei Knoten, kann ein Angreifer aber in der Realität die genaue Verteilung nicht kennen.

Wie zu erwarten war, erhält man eine geringere Entropie als bei der empirischen Entropie. Dennoch ist in beiden Schätzungen der Trend der Entropie - und somit auch der Sicherheit - im Zusammenhang mit dem Securitiy Indicator zu erkennen: Je niedriger der SI, desto niedriger auch die Sicherheit. Schlüssel mit einem SI $< 0,5$ sollten verworfen werden, da ihre Entropie sehr gering ist. Man erkennt dies auch schon mit dem bloßen Auge bei der Betrachtung von Schlüsseln mit einem kleinen SI. Sie enthalten aufgrund der Kodierung sehr viele 0en und bestehen bei einem Großteil der Schlüssel mit einem SI $< 0,1$ ausschließlich aus 0en. Wie man aber weiter an den Zahlen der Tabellen sieht, sind die Schlüssel mit einem derart niedrigen SI relativ gering ($13,5\%$ im Experiment). Diese „schlechten“ Schlüssel sind häufig auf temporäre

Interferenzen zurückzuführen und bei einem wiederholten Schlüsselaustausch nach wenigen Sekunden konnte nahezu jedesmal ein „guter“ Schlüssel generiert werden, ohne die Position zu ändern.

Schwachstellen/Mögliche Agriffstechniken

TinyOS-Anwendungen für Sensorknoten sollten beim Verwenden der `KeyGenC`-Komponente demnach *immer* den SI der generierten Schlüssel beachten. Dennoch hat die momentane Implementierung des Protokolls Schwachstellen. So bietet beispielsweise das Toleranzwahlverfahren eine gute Angriffsstelle. Bei der Suche nach der maximalen Schwankung wird die Toleranz für alle Kanäle schrittweise erhöht. Bei jedem dieser Schritte wird die MD5-Summe zu den korrigierten Werten übertragen. Eve kennt nun sowohl diese Summe als auch die dazugehörige Toleranz mit welcher die Werte korrigiert wurden. Im letzten Schritt der Suche, bei welchem alle gemessenen RSS-Werte mit t_{max} korrigiert werden, kann Eve den Klartext zur MD5-Summe mit dem Aufwand, der bei einem SI von t_0 / t_{max} nötig wäre, berechnen. Somit hätte sie mindestens ein Byte des Schlüssels herausgefunden, nämlich genau die Bytes, welche zu Kanälen mit einer Schwankung von t_{max} gehören. Sie weiß zwar nicht, zu welchen Kanälen dieser Wert gehört, da aber die Unterschiede in der RSS unter den Kanälen nicht allzu groß sind, kann Eve den Schlüssel unter Umständen mit relativ geringem Aufwand finden. Im oben erwähnten Beispielexperiment mit 9 MICAz-Knoten lag die durchschnittliche Differenz zwischen der größten und der kleinsten gemessenen Signalstärke auf den 16 verfügbaren Kanälen bei 16 dB. Das bedeutet, dass die Messungen der Kanäle im Durchschnitt in einem Fenster von 16 dB lagen und Eve die Messungen von Alice und Bob nur innerhalb dieses Fensters um den aufgedeckten Wert suchen muss.

Ist Eve in der Lage Interferenzen gezielt zu erzeugen, könnte sie den Verlauf des Protokolls gezielt stören. So könnte sie beispielsweise die Messungen von Alice auf einem bestimmten Kanal so stören, dass hohe Schwankungen auftreten. Das Toleranzwahlverfahren wäre dann gezwungen in der ersten Phase die Toleranzen bis auf diese hohe Schwankung zu erhöhen. Würde Eve es dann schaffen, aus der übertragenen MD5-Summe das Byte des Kanals mit der höchsten Schwankung zu errechnen, könnte sie dieses dem Kanal zuordnen, da sie die Schwankungen selbst erzeugt hat. War die Schwankung hoch genug, müsste Alice das Byte auf $\texttt{RSSI}_{\texttt{min}}$ „runterkorrigieren“ und Eve könnte mit Hilfe des übertragenen Anpassungsvektors Bob's genauen RSS-Wert berechnen. Das folgende Beispielsszenario soll diese Angriffstechnik noch einmal verdeutlichen.

Beispiel: Alice und Bob versuchen das geteilte Geheimnis aus ihrem Funkkanal zu extrahieren. Eve belauscht den Ablauf des Protokolls und stört Alice's Messungen auf Kanal 16. Angenommen Bob's und Alice's Ergebnisse der Sampling Phase wären dann folgende Vektoren:

Kanal	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Alice	-50,1	-52,5	-51,9	-55,0	-58,3	-85,1	-65,3	-65,6	-63,7	-60,2	-59,8	-55,3	-52,0	-54,0	-55,8	-54,3
Bob	-50,3	-53,0	-51,8	-55,0	-58,7	-61,3	-65,6	-66,0	-64,1	-60,4	-59,7	-55,3	-52,1	-54,3	-55,2	-55,1

Die Differenz auf Kanal 16 beträgt $|(-85,1) - (-61,3)| = 23,8$ dB. Das Toleranzwahlverfahren erhöht in der ersten Phase die Toleranz also auf 24, um diese Schwankung auszugleichen (Schrittweite von 0,25). Nach Gleichung 3.2 korrigiert Bob alle seine Messungen auf -95 dBm. Die Werte, die er dann Alice im Anpassungsvektor für diese Key Generation Phase sendet, entsprechen dann genau der Differenz von -95 und Bob's Messung. Auf diese Weise wüsste Eve alle Messwerte von Bob und könnte mit wenigen Versuchen auch die von Alice herausbekommen. Da die Schwankung aber nur auf einem Kanal auftritt und alle anderen Kanäle mit der Standardtoleranz von 0,5 korrigiert werden können, hat der Schlüssel trotz dass er aufgedeckt wurde noch einen SI von immerhin 0,25.

Eine weitere mögliche Angriffsstelle für einen feindlichen Lauscher stellen die übertragenen MD5-Summen dar. Da die Schlüssellänge immer gleich bleibt und die Anzahl der möglichen Schlüssel relativ begrenzt ist, könnte auf Dauer eine Datenbank angelegt werden um erfolgreich *Reverse Lookups* der MD5-Summen durchzuführen und so den Schlüssel herauszufinden.

5.1.3 Kalibrierung

Um die in Kapitel 4 vorgestellte Kalibrierung im Einsatz mit dem Schlüsselaustauschprotokoll zu testen, wurden zwei Sensorknoten gewählt, die eine starke Abweichung in den RSSI-Werten aufweisen. Zuerst wurden 10 Schlüssel von zufälligen Positionen generiert. Danach wurden die beiden Knoten mit jeweils 50 Samples pro Kanal durch die `CalibrationApp`-Anwendung kalibriert. Ein erneutes Generieren von 10 Schlüsseln aus zufälligen Positionen hat eine extreme Verbesserung des SI gezeigt: Die Schlüssel vor der Kalibrierung hatten im Schnitt einen SI von 0,03 und bestanden alle nur aus 0en. Nach der Kalibrierung war der durchschnittliche SI 0,77, wobei der SI von nur einem Schlüssel unter 0,5 war.

Da der SI direkt aus den verwendeten Toleranzen berechnet wird und diese wiederum ein Maß für die Schwankungen sind, zeigt eine derartige Verbesserung des SI, dass die Schwankungen durch die Kalibrierung extrem reduziert wurden. Im Mittel wurden die Schwankungen bei diesem Experiment auf weniger als 1 dB reduziert.

6 Zusammenfassung und Ausblick

In dieser Arbeit wurde die Implementierung eines Schlüsselaustauschprotokoll für TinyOS vorgestellt. Die Schlüsselgenerierung basiert hier nicht auf mathematischen Verfahren, wie viele andere Vertreter dieser Kategorie von Protokollen. Das hier implementierte Verfahren nutzt die Unvorhersagbarkeit der physikalischen Eigenschaften des Funkkanals und das Prinzip der Reziprozität aus, um zwei Knoten mit einem gemeinsamen Geheimnis zu versorgen. Da bei diesem Verfahren keine aufwendigen Berechnungen nötig sind, ist es besonders für den Einsatz in WSNs geeignet, da diese im Allgemeinen eine relativ schwache Rechenleistung aufweisen. Mit einem generierten Schlüssel mit einer Breite von 128 bit als Ergebnis, eignet sich das Protokoll beispielsweise sehr als Ergänzung zum Einsatz des *Advances Encryption Standard* (AES) zum Verschlüsseln von Daten. AES wird für eine Schlüssellänge von 128 Bit beispielsweise auf MICAz-Knoten direkt von der Hardware unterstützt [15].

Die Anforderungen der in dieser Arbeit implementierten TinyOS-Komponente an den Sensorknoten sind folgende:

- Unterstützung der Plattform durch TinyOS
- Verwendung des CC2420-Chips zur drahtlosen Kommunikation
- Fließkommazahlenunterstützung (durch Software oder Hardware)

Das Protokoll kann aber prinzipiell auf jeder Plattform implementiert werden, die einen Chip verwendet, der die empfangene Signalstärke messen kann. Unter einem Verlust an Sicherheit kann auch auf die Fließkommarithmetik verzichtet werden. Somit kann die Komponente relativ einfach auf andere Plattformen übertragen werden.

Da der hier verwendete CC2420-Chip bei der Messung der RSS leider nur eine Genauigkeit von ± 6 dB bietet, können sich die Knoten in der Wahrnehmungen von Signalstärken unterscheiden [14]. Da eine einheitliche Signalmessung aber signifikant für den Erfolg des Schlüsselaustauschprotokolls ist, wurde zusätzlich eine grobe Kalibrierung implementiert um Knoten in ihrer Wahrnehmung anzupassen. Wie Experimente gezeigt haben, kann durch den Einsatz dieser Kalibrierung in einzelnen Fällen eine erhebliche Verbesserung der Sicherheit des Protokolls erzielt werden. Auf Grund der relativ hohen Genauigkeit des RSSI ist in den meisten Fällen keine Kalibrierung nötig. Um eine große Anzahl an Sensorknoten zu kalibrieren, könnte das Verfahren so erweitert werden, dass die Knoten untereinander auch Kalibrierungsinformationen über andere Knoten austauschen können.

Trotz Reziprozität und Kalibrierung kommt es aufgrund der Antisymmetrie des Funkkanals immer noch zu Differenzen in der Wahrnehmung des Kanals. Aus diesem Grund verwendet das Protokoll zusätzlich eine Fehlerkorrektur. Die im Protokoll zur Fehlerkorrektur verwendeten Toleranzen haben - wie gezeigt wurde - direkte Auswirkungen auf die Sicherheit des Schlüssels. Um einen Ausgleich zwischen Sicherheit und Nachrichtenaufwand zu schaffen und die größt mögliche Entropie aus dem gemeinsamen Geheimnis zu schöpfen, wurde ein Toleranzwahlverfahren implementiert, welches die verwendeten Toleranzen dynamisch auf die Schwankungen in den Messungen anpassen soll.

Um die Flexibilität der Implementierung zu erhöhen, könnte die Möglichkeit der Auswahl eines Toleranzwahlverfahrens geboten werden. So könnte zum Beispiel für ein sicherheitskritisches Szenario, bei welchem der Kommunikationsaufwand eine weniger wichtige Rolle als die Sicherheit spielt, ein feineres Toleranzwahlverfahren ausgewählt werden, welches zwar eine maximal mögliche Sicherheit bietet, dafür aber viele Nachrichten austauschen muss. Für Szenarien, bei denen ein Schlüssel nur kurzfristig benötigt wird (z.B. bei einem Authentifizierungsverfahren) und eine schnelle Terminierung wichtig ist, könnte ein groberes Toleranzwahlverfahren gewählt werden.

Der *Security Indicator* (SI) eines Schlüssels bietet einer Anwendung Informationen über die Sicherheit des generierten Schlüssels. Dies ist vor allem für sicherheitskritische Anwendungen wichtig. Typische Sensornetzwerke bestehen heute aus hunderten oder tausenden Sensorknoten. Diese Sensorknoten können nur direkt mit anderen Sensorknoten oder mit einer externen Basisstation kommunizieren. Um nun Daten von überall aus dem WSN zur Basisstation zu senden, ist der Einsatz von *Multi-Hop Routing-Algorithmen* nötig, da sie über mehrere Knoten zur Basisstation geleitet werden müssen. Wie Al-Karaki und Kamal in [2] festgestellt haben, wird Sicherheit in bisherigen WSN-Routing-Algorithmen oftmals nicht beachtet. Für den Austausch von geheimen Daten spielt sie aber eine große Rolle. Mit dem hier vorgestellten Schlüsselaustausch-Protokoll könnten Schlüssel ad-hoc für jeden Link in der Topologie erzeugt werden und mit Hilfe des SI ein geeignetes Routing-Protokoll entworfen werden, was den SI beachtet und so größt mögliche Sicherheit auf der Multi-Hop Route vom Sensorknoten zur Basisstation garantiert.

Die Berechnung des SI sollte jedoch noch verfeinert werden, um die Anwendung auf Sicherheitsrisiken, wie sie in Abschnitt 5.1.2 geschildert werden, hinzuweisen. Die Evaluierung des Verfahrens hat gezeigt, dass Schlüssel mit einem SI von unter 0,5 auf Grund mangelnder Sicherheit im Allgemeinen verworfen werden sollten. Um die erwähnten Sicherheitheitsbedenken bezüglich der Verwendung von Hashfunktionen in der Acceptance Phase zu beseitigen, können für das Verifizieren des gemeinsamen Geheimnisses alternative *Challenge-Response-Verfahren* eingesetzt werden.

Literaturverzeichnis

[1] Make. Website. `http://www.gnu.org/software/make/`.

[2] J. N. Al-Karaki and A. E. Kamal. Routing techniques in wireless sensor networks: a survey. *IEEE Wireless Communications*, 11(6):6–28, 2004.

[3] J. B. Andersen, T. S. Rappaport, and S. Yoshida. Propagation measurements and models for wireless communications channels. *Communications Magazine, IEEE*, 33(1):42–49, 1995.

[4] Georgia E. Athanasiadou and Andrew R. Nix. A novel 3-d indoor ray-tracing propagation model: the pathgenerator and evaluation of narrow-band and wide-band predictions. *IEEE Transactions on Vehicular Technology*, 49(4):1152–1168, Jul 2000.

[5] Babak Azimi-Sadjadi, Aggelos Kiayias, Alejandra Mercado, and Bulent Yener. Robust key generation from signal envelopes in wireless networks. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 401–410, New York, NY, USA, 2007. ACM.

[6] Haowen Chan, Adrian Perrig, and Dawn Song. Random key predistribution schemes for sensor networks. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 197, Washington, DC, USA, 2003. IEEE Computer Society.

[7] Laurent Eschenauer and Virgil D. Gligor. A key-management scheme for distributed sensor networks. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 41–47, New York, NY, USA, 2002. ACM.

[8] David Gay, Phil Levis, and David Culler. Software design patterns for tinyos. *SIGPLAN Not.*, 40(7):40–49, 2005.

[9] David Gay, Matt Welsh, Philip Levis, Eric Brewer, Robert von Behren, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 1–11, 2003.

[10] Peter Grünwald and Paul M. B. Vitányi. Shannon information and kolmogorov complexity. *CoRR*, cs.IT/0410002, 2004.

[11] Panu Hämäläinen, Mauri Kuorilehto, Timo Alho, Marko Hännikäinen, and Timo D. Hämäläinen. Security in wireless sensor networks: Considerations and experiments. In *SAMOS*, pages 167–177, 2006.

[12] Carl Hartung, James Balasalle, Richard Han, Carl Hartung, James Balasalle, and Richard Han. Node compromise in sensor networks: The need for secure systems. Technical report, Dept of Comp Sci, Univ of Colorado at Boulder, 2005.

[13] IEEE 802 LAN/MAN Standards Committee. Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs). Technical report, The Institute of Electrical and Electronics Engineers, Inc. , June 2006. IEEE Std 802.15.4™-2006.

[14] Chipcon Inc. CC2420 Data Sheet. Online - zugegriffen am 3. Januar 2010. `http://www.ti.com/lit/gpn/cc2420`.

[15] Crossbow Inc. MICAz Data Sheet. Online - zugegriffen am 3. Januar 2010. `http://www.xbow.com`.

[16] M.F. Iskander and Zhengqing Yun. Propagation prediction models for wireless communication systems. *IEEE Transactions on Microwave Theory and Techniques*, 50(3):662–673, Mar 2002.

[17] Suman Jana, Sriram Nandha Premnath, Mike Clark, Sneha K. Kasera, Neal Patwari, and Srikanth V. Krishnamurthy. On the effectiveness of secret key extraction from wireless signal strength in real environments. In *MobiCom '09: Proceedings of the 15th annual international conference on Mobile computing and networking*, pages 321–332, New York, NY, USA, 2009. ACM.

[18] Philip Levis. TEP 111: message_t. TinyOS 2.0 Documentation. `http://www.tinyos.net/tinyos-2.x/doc/`.

[19] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. Tinyos: An operating system for sensor networks. In *Ambient Intelligence*. Springer Verlag, 2004.

[20] Ming Li and Paul M.B. Vitnyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer Publishing Company, Incorporated, 2008.

[21] An Liu and Peng Ning. TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks. In *7th International Conference on Information Processing in Sensor Networks (IPSN 2008)*, pages 245–256, April 2008.

[22] Ruoshui Liu and Ian J. Wassell. A novel auto-calibration system for wireless sensor motes. Technical Report UCAM-CL-TR-727, University of Cambridge, Computer Laboratory, Sep 2008.

[23] Suhas Mathur, Wade Trappe, Narayan Mandayam, Chunxuan Ye, and Alex Reznik. Radio-telepathy: extracting a secret key from an unauthenticated wireless channel. In *MobiCom '08: Proceedings of the 14th ACM international conference*

on Mobile computing and networking, pages 128–139, New York, NY, USA, 2008. ACM.

[24] Ueli M. Maurer and Stefan Wolf. The relationship between breaking the diffie–hellman protocol and computing discrete logarithms. *SIAM J. Comput.*, 28(5):1689–1721, 1999.

[25] Adrian Perrig, Robert Szewczyk, Victor Wen, David Culler, and J. D. Tygar. Spins: Security protocols for sensor networks. In *Wireless Networks*, pages 189–199, 2001.

[26] Ronald L. Rivest. The MD5 Message-Digest Algorithm (RFC 1321). `http://www.ietf.org/rfc/rfc1321.txt?number=1321`, April 1992.

[27] M. J. B. Robshaw. On recent results for MD2, MD4 and MD5. *RSA Laboratories' Bulletin*, 4, 1996.

[28] Günter Schäfer. *Netzsicherheit: Algorithmische Grundlagen und Protokolle*. dpunkt.verlag, 2003.

[29] Jochen Schiller. *Mobilkommunikation*. Addison-Wesley, 1 edition, 2000.

[30] Thomas Schürmann and Peter Grassberger. Entropy estimation of symbol sequences. 2002.

[31] Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 625–56, Jul, Oct 1948.

[32] Ulrich Speidel and Mark Titchener. How well do practical information measures estimate the shannon entropy. In *Proceedings of the 5th International Symposium on Communication Systems and Digital Signal Processing (CSNDSP) 2006*, 2006.

[33] Kannan Srinivasan and Philip Levis. RSSI is Under Appreciated. In *Proceedings of the Third Workshop on Embedded Networkde Sensors (EmNets)*, May 2006.

[34] Piotr Szczechowiak, Leonardo B. Oliveira, Michael Scott, Martin Collier, and Ricardo Dahab. Nanoecc: Testing the limits of elliptic curve cryptography in sensor newtorks. In *EWSN*, volume 4913 of *Lecture Notes in Computer Science*, pages 305–320. Springer, 2008.

[35] Mark R. Titchener, Radu Nicolescu, Ludwig Staiger, Aaron Gulliver, and Ulrich Speidel. Deterministic complexity and entropy. *Fundam. Inf.*, 64(1-4):443–461, 2004.

[36] Ronald Watro, Derrick Kong, Sue-fen Cuti, Charles Gardiner, Charles Lynn, and Peter Kruus. Tinypk: securing sensor networks with public key technology. In *SASN '04: Proceedings of the 2nd ACM workshop on Security of ad hoc and sensor networks*, pages 59–64, New York, NY, USA, 2004. ACM.

[37] Matthias Stephan Wilhelm. Implementation and analysis of a key generation protocol for wireless sensor networks. Diplomarbeit, Technische Universität Kaiserslautern, 2009.