Key Distribution Schemes for Resource-Constrained Devices in Wireless Sensor Networks

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik
der Universität Stuttgart zur Erlangung der Würde eines
Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

vorgelegt von

Arno Rüdiger Wacker

aus Temeschburg

Hauptberichter: Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel

Mitberichter: Prof. Dr.-Ing. Torben Weis

Tag der mündlichen Prüfung: 15.08.2007

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart

2007

Abstract

Wireless sensor networks based on highly resource-constrained devices require symmetric cryptography in order to make them secure. Integral to this is the exchange of unique symmetric keys between two devices. In this dissertation, we propose three novel decentralized key distribution schemes that guarantee the confidentiality of a key exchange even if an attacker has compromised some of the devices in the network.

Our first key distribution scheme – the basic key distribution scheme – guarantees the confidentiality of any new established key in case there are only eavesdropping attacker and no device failures present. Our second scheme – the fault-tolerant key distribution scheme – extends the basic scheme so that also more powerful attackers and device failures can be handled. Our third proposed key distribution scheme – the extended key distribution scheme – is also based on the basic scheme but further optimized in terms of memory consumption and network traffic.

A central objective of all key distribution scheme designs was to minimize resource consumption on the individual devices. We evaluate the resource requirements of our schemes in terms of attacker resilience, memory requirements, and network traffic both through theoretical analysis and through simulations.

Zusammenfassung

Drahtlose Sensornetze gewannen in den letzen Jahren in den verschiedensten Anwendungsgebieten an Bedeutung. Sie werden z.B. für die Erkennung von Waldbränden [DS05], die Beobachtung von Wildtieren in ihrer natürlichen Umgebung [MCP+02] und in der Landwirtschaft [BBB04] verwendet. Des Weiteren trifft man drahtlose Sensornetze auch in verschiedene Szenarien des *Pervasive Computings* [Wei91] an. Das gängigste Beispiel hierfür ist Heimautomatisierung bzw. Gebäudemanagement: Um den individuellen Wohnkomfort zu erhöhen, ist in diesem Beispiel ein Gebäude, z.B. ein privates Heim, mit einer großen Menge an Sensoren und Aktoren ausgestattet. Dadurch kann, z.B. mit Hilfe der Sensordaten und Steuermöglichkeit der Aktoren, die Heizung automatisch eingeschaltet werden, sobald der/die Besitzer(in) sich auf den Heimweg begibt. Sobald er/sie das Haus erreicht, wird er/sie von Sensoren erkannt (und authentifiziert) worauf sich die Tür automatisch für ihn oder sie öffnet. Falls es in einem Raum zu dunkel ist, wird automatisch das Licht eingeschaltet sobald dieser Raum betreten wird.

Sicherheit ist ein wesentliches Kriterium bei der Entwicklung von Sensornetzen. Falsche oder manipulierte Sensorwerte (z.B. durch Vandalismus) stellen ein großes Problem für Überwachungsnetze dar. Wenn, z.B. durch manipulierte Daten, häufig ein falscher Alarm für einen Waldbrand ausgelöst wird, so führt das zu unnötigen Einsätzen der Feuerwehr und letztlich zur Abschaltung des Systems. Auch beim Gebäudemanagement ist Sicherheit ein wichtiges Kriterium, da es durch diese Technologie neue Möglichkeiten zur Verletzung der Privatsphäre des Einzelnen gibt. Hacker können durch Auslesen der Sensordaten private Personen in ihrem eigenen Heim beobachten. Ein potentieller Dieb kann eventuell herausfinden ob jemand zu Hause ist. Eine moderne Form des Vandalismus wäre z.B. das Abschalten der Heizung im Winter. All diese Beispiele zeigen, dass Sicherheit in drahtlosen Sensornetzen ein wichtiges Kriterium für die Akzeptanz dieser Technologie ist. Sensordaten müssen so verschlüsselt werden, dass nur autorisierte Personen diese lesen können bzw. allgemein der Zugang zu Sensoren und Aktoren muss so eingeschränkt werden, dass ein Missbrauch davon unmöglich ist. Das Einspeisen von manipulierten Daten in solch ein Netz darf nicht möglich sein.

Das Bereitstellen einer geeigneten Sicherheitslösung ist keine einfache Aufgabe. Im Allgemeinen ist es nicht möglich klassische Sicherheitslösungen zu verwenden da diese Lösungen meist bei starken Ressourcenbeschränkungen, wie man sie bei Geräten aus Sensornetze antrifft, nicht

anwendbar sind. Daher müssen neue, spezialisierte Lösungen entwickelt werden, welche mit den vorhandenen und stark begrenzten Ressourcen auskommen.

Für die Bereitstellung von Sicherheit in drahtlosen Sensornetzen müssen eine Reihe von Sicherheitsmechanismen angewendet werden. Die Grundlage dieser Mechanismen bilden kryptografische Verschlüsselungen. Mit Hilfe von kryptografischer Verschlüsselung können die meisten Sicherheitsziele, wie z.B. Authentifikation, Geheimhaltung und Datenintegrität, erreicht werden. Bei modernen kryptografischen Verfahren beruht die Sicherheit der Verschlüsselung auf der Verwendung von *Schlüsseln*. Schlüssel sind Bitsequenzen einer angemessenen Länge welche verwendet werden um Daten zu verschlüsseln. Nur der Besitzer des entsprechenden Schlüssels kann die verschlüsselten Daten entschlüsseln – die Entschlüsselung ohne Kenntnis des entsprechenden Schlüssels ist unmöglich, soweit das verwendete kryptografische Verfahren sicher ist. Im Allgemeinen können kryptografische Verfahren in *symmetrische* und *asymmetrische* Algorithmen eingeteilt werden (z.B. [Sch95]). Konzeptionell verwendet man bei symmetrischen Verfahren den gleichen Schlüssel zur Ver- und Entschlüsselung. Dies bedeutet, dass der Schlüssel den beiden Kommunikationspartner bekannt sein muss bevor sie Daten sicher austauschen können. Im Gegensatz hierzu benutzt man bei asymmetrischen Algorithmen zwei verschiedene Schlüssel – einen zum Ver- und den Anderen zum Entschlüsseln.

Bevor eine sichere Kommunikation zwischen zwei Kommunikationspartnern stattfinden kann, muss ein sicherer Schlüsselaustausch zwischen den Partnern erfolgen. Hierzu wird ein für drahtlose Sensornetze geeignetes Schlüsselaustauschverfahren benötigt. Im Allgemeinen kann man Schlüsselaustauschverfahren anhand von zwei Dimensionen klassifizieren. Die erste Dimension ist die Art des verwendeten kryptografischen Algorithmen, d.h. ob symmetrische oder asymmetrische Algorithmen verwendet werden. Die zweite Dimension ist die Systemorganisation. Hier kann zwischen einem zentralisierten und einem dezentralisiertem Ansatz unterschieden werden. Ein Schlüsselaustauschverfahren kann entweder eine zentrale Stelle, z.B. einen Authentifizierungsserver verwenden oder vollkommen dezentral arbeiten. Diese Klassifizierung führt zu vier verschiedene Klassen von Schlüsselaustauschverfahren, die im Folgenden diskutiert und bewertet werden.

Existierende Schlüsselaustauschverfahren

Die einfachste Möglichkeit einen sicheren Schlüsselaustausch zu realisieren ist die Verwendung einer zentralen Stelle, eines sogenannten Authentifizierungsservers [PK79]. Dafür muss jedes Gerät einen gemeinsamen (geheimen) Schlüssel mit dem Authentifizierungsserver besitzen. Damit ein Gerät mit einem anderen Gerät im Netz sicher kommunizieren kann, wird der Authentifizierungsserver angefragt, welcher daraufhin einen Schlüssel zwischen den beiden Kommunikationspartner vermittelt. Mit dem vom Authentifizierungsserver generierten Schlüs-

sel können die beiden Kommunikationspartner sicher kommunizieren. Bekannte Beispiele die diesen Ansatz verwenden sind [KN93, NS78]. Die Verwendung eines zentralen Authentifizierungsservers hat allerdings verschiedene Nachteile. Ein erfolgreicher Angriff auf den Authentifizierungsserver führt zur Kompromittierung des gesamten Netzes. Des Weiteren ist bei einem Ausfall des zentralen Servers keine sichere Kommunikation im Netz mehr möglich.

Dezentrale Ansätze vermeiden die oben geschilderten Nachteile, da sie keinen zentralen Server im Netz benötigen. Stattdessen kooperieren mehrere Teilnehmer des Netzes miteinander um neue Schlüssel zu erstellen und zu verteilen. Dezentrale Ansätze können sowohl auf asymmetrischer Kryptographie (z.B. [ZH99, HBC01]) als auch auf symmetrischer Kryptographie (z.B. [CPS03, ZXSJ03, CP05]) basieren. Das erste dezentrale Schlüsselaustauschverfahren, das nur auf symmetrischer Kryptographie basiert, wurde im Jahre 2003 in [CPS03] vorgeschlagen und basiert auf einer Idee von Gong [Gon93]. Die Grundidee von Gongs Ansatz ist es, ein gemeinsames Geheimnis unter einer bestimmten Anzahl von Teilnehmern aufzuteilen, wodurch niemals ein einzelner in den Besitz des gesamten Geheimnisse kommt. Diese Idee wurde von Chan et al. für drahtlose Sensornetze angepasst. Hier wird der Schlüssel, der sicher zwischen zwei Kommunikationspartnern ausgetauscht werden soll, in mehrere Teile zerlegt. Damit ein Angreifer diesen Schlüssel kompromittieren kann, muss er sich Zugang zu allen Teilen des Schlüssels verschaffen. Zwei Teilnehmer (Geräte) eines drahtlosen Sensornetzes, die noch keinen gemeinsamen Schlüssel besitzen, können einen neuen Schlüssel auf folgende Art austauschen: Ein Gerät generiert eine bestimmte Anzahl von Teilschlüsseln und sendet diese dann auf unterschiedlichen Pfaden durch das Netzwerk zum Kommunikationspartner. Auf jedem Hop wird der Teilschlüssel mit dem jeweils gültigen Schlüssel gesichert. Beide Kommunikationspartner generieren nun den endgültigen Schlüssel durch die bitweise XOR-Verknüpfung aller Teilschlüssel. Dieser Ansatz funktioniert allerdings nur, wenn es zwischen den Kommunikationspartnern genügend unterschiedliche, sogenannte knotendisjunkte, Pfade gibt. Ist dies nicht gegeben so kann kein neuer Schlüssel ausgetauscht werden. Keiner der bislang existierenden Ansätze kann die Existenz dieser Pfade garantieren und somit auch nicht, dass ein Schlüsselaustausch zwischen beliebigen Geräten möglich ist [CPS03, ZXSJ03].

In dieser Arbeit werden drei neue dezentrale Schlüsselaustauschverfahren vorgestellt. Diese basieren nur auf symmetrischer Kryptographie und garantieren den Schlüsselaustausch zwischen beliebigen Teilnehmern des Netzes. Dies wird erreicht, indem der Ansatz garantiert, dass es immer ausreichend viele knotendisjunkte Pfade zwischen beliebigen Geräten gibt.

Anforderungen an ein Schlüsselaustauschverfahren

In der Vergangenheit wurden in der Literatur eine Reihe von Schlüsselaustauschverfahren für verschiedene Anwendungsgebiete wie z.B. Ad-Hoc Netze [ZH99] oder dem Internet [KN93]

vorgestellt. Ein Schlüsselaustauschverfahren für drahtlose Sensornetze bringt allerdings neue Herausforderungen mit sich. In diesem Abschnitt werden diese Herausforderungen analysiert und eine Reihe von Anforderungen an ein Schlüsselaustauschverfahren für drahtlose Sensornetze ausgearbeitet.

Garantierter Schlüsselaustausch

Bevor zwei Geräte eines drahtlosen Sensornetzes sicher miteinander kommunizieren können, müssen diese zunächst einen gemeinsamen Schlüssel austauschen. Ein Schlüsselaustauschverfahren muss garantieren, dass dies zu jedem Zeitpunkt möglich ist. Es ist nicht akzeptabel, dass zwei beliebige Geräte zu irgendeinem Zeitpunkt keinen gemeinsamen Schlüssel aufbauen können.

Verwendung symmetrischer Kryptographie

Für die Bereitstellung von Sicherheit in drahtlosen Sensornetzen benötigt man kryptografische Mechanismen welche Geheimhaltung, Authentizität, Integrität und Frische der Nachrichten garantieren. Diese Mechanismen können sowohl mit symmetrischer als auch mit asymmetrischer Kryptographie realisiert werden. Die eingesetzte Klasse von kryptografischen Algorithmen hängt unter anderem maßgeblich von den zur Verfügung stehenden Ressourcen auf den eingesetzten Geräten ab. Asymmetrische Kryptographie benötigt eine Größenordnung mehr an Ressourcen gegenüber symmetrischer Kryptographie [CKM00], die auch auf ressourcenschwachen Geräten effizient implementiert werden kann. Da in dieser Dissertation von solchen Geräten ausgegangen wird, muss das zu entwickelnde Schlüsselaustauschverfahren ausschließlich auf symmetrischer Kryptographie basieren.

Schrittweise Verminderung der Sicherheit

Üblicherweise sind die in drahtlosen Sensornetzen eingesetzten Geräte sehr preisgünstig und sehr klein. Daher kann man nicht erwarten, dass sie einem physischen Einbruchsversuch standhalten (engl. *tamper resistant*). Daher müssen wir annehmen, dass es einem Angreifer gelingen wird, einige Geräte unter seine Kontrolle zu bringen. Dies muss bei der Entwicklung eines Schlüsselaustauschverfahrens berücksichtigt werden. Das Verfahren muss trotz einiger vom Angreifer kontrollierter Geräte noch funktionsfähig sein. In der Literatur wird dies als schrittweise Verminderung (engl. *graceful degradation*) bezeichnet. Um dies zu erreichen muss ein Schlüsselaustauschverfahren zwei weiteren Anforderungen genügen: Erstens, es müssen paarweise eindeutige Schlüssel verwendet werden, da sonst die Übernahme eines einzelnen Gerätes

zum Verlust des gesamten Netzes führen würde. Zweitens, darf das Netz keine ausgezeichneten Geräte enthalten, welche z.B. als zentraler Authentifizierungsserver arbeiten. Durch die Übernahme diese Gerätes wäre das gesamte Netz kompromittiert. Daher muss das eingesetzte Schlüsselaustauschverfahren vollständig dezentral arbeiten und keine zentralisierten Komponenten verwenden.

Dynamische Netzgröße

In vielen Szenarios, in denen drahtlose Sensornetze eingesetzt werden, kann die Größe des Netzes nicht vorherbestimmt werden. Wird z.B. ein Sensornetz für das Gebäudemanagement verwendet, so muss man dem Benutzer die Möglichkeit geben, einzelne Geräte später dem Netz hinzuzufügen bzw. alte Geräte aus dem Netz zu entfernen. Dies bedeutet dass ein Schlüsselaustauschverfahren eine dynamisch wechselnde Netzgröße unterstützen muss.

Grundlegendes Schlüsselaustauschverfahren

Für die Erfüllung aller aufgestellter Anforderungen wurden im Rahmen dieser Dissertation drei Schlüsselaustauschverfahren entwickelt. Alle drei vorgestellten Verfahren basieren auf dem Basisverfahren welches im Folgenden kurz vorgestellt wird.

Wie bereits erwähnt, werden zum Erreichen der schrittweisen Verminderung der Sicherheit paarweise Schlüssel benötigt. Durch die Ressourcenbeschränkung der Geräte ist es üblicherweise nicht möglich paarweise Schlüssel zwischen allen Gerätepaaren auf den Geräten zu speichern. Daher wird nur eine Teilmenge von Schlüssel auf den einzelnen Geräten gespeichert. Dies führt zu zwei Fragestellungen: Welche Schlüssel sollen auf welchen Geräten gespeichert werden und wie können Schlüssel zwischen zwei Geräten zur Laufzeit aufgebaut werden, wenn solch ein Schlüssel noch nicht existiert. Ein typisches Schlüsselaustauschverfahren besteht daher aus zwei Teilen, nämlich der Schlüsselvorverteilung (oder auch dem Netzaufbau) und dem Schlüsselaustauschprotokoll.

Die Schlüsselvorverteilung ist zuständig für die Auswahl der initialen Teilmengen von Schlüsseln, die auf den Geräten gespeichert werden. Dies wird üblicherweise über einen externen sicheren Kanal durchgeführt. Als Beispiel kann der Hersteller eines drahtlosen Sensornetzes die initiale Teilmenge von Schlüsseln auf den Geräten speichern. Dies geschieht üblicherweise zur Produktionszeit und man kann davon ausgehen, dass dies in einer sicheren Umgebung stattfindet, d.h. es ist kein Angreifer anwesend.

Unter Zuhilfenahme der Schlüssel welche auf den Geräten durch die Schlüsselvorverteilung gespeichert wurden, ist es mit dem Schlüsselaustauschprotokoll möglich weitere, noch nicht

existierende Schlüssel, zu erstellen. Im Gegensatz zur Schlüsselvorverteilung wird das Schlüsselaustauschprotokoll zur Laufzeit ausgeführt, wodurch die Anwesenheit eines Angreifers nicht mehr ausgeschlossen werden kann.

Im Folgenden werden zunächst einige Begriffe eingeführt und danach die Algorithmen für die Schlüsselvorverteilung und das Schlüsselaustauschprotokoll vorgestellt.

Parameter und Bezeichnungen

Für die Beschreibung des Schlüsselaustauschverfahrens werden in dieser Dissertation einige Begriffe und Parameter verwendet, welche im Folgenden beschrieben werden. Das Sensornetz wird als ein ungerichteter Graph dargestellt, wobei die Knoten des Graphen die einzelnen Geräte repräsentieren. Die Kanten des Graphen repräsentieren einen symmetrischen Schlüssel zwischen den beiden entsprechenden Geräten. Dieser Graph wird im Rahmen dieser Arbeit als *Schlüsselgraph* bezeichnet. Des Weiteren wird der Begriff *Sicherheitsstufe* eingeführt. Die Sicherheitsstufe s des Verfahrens ist ein frei wählbarer Parameter welcher die Widerstandsfähigkeit des Netzes gegenüber dem Angreifer steuert. In dem vorgestellten Verfahren wird *garantiert*, dass das Netz absolut sicher ist, solange der Angreifer nicht mindestens s Geräte unter seine Kontrolle bringen konnte.

Übersicht

Die neue und zugrundeliegende Idee des hier vorgestellten Schlüsselaustauschverfahrens ist es, zu garantieren, dass man mindestens einen s-verbundenen Schlüsselgraphen erzeugt. Ein Graph ist genau dann s-verbunden wenn man eine beliebige Teilmenge von weniger als s Knoten aus dem Graphen entfernen kann, ohne dass dieser Graph dadurch zerfällt. Der Schlüssevorverteilungsalgorithmus konstruiert einen solchen s-verbunden Graphen. Man kann zeigen, dass es in einem s-verbundenen Graphen zwischen beliebigen, nicht-benachbarten Knoten mindestens s knotendisjunkte Pfade gibt [Men27]. Mit Hilfe dieser Pfade ist das Schlüsselaustauschprotokoll in der Lage auf sichere Weise einen neuen Schlüssel zwischen zwei Geräten auszutauschen.

Schlüsselvorverteilung – Netzaufbau

Der Schlüsselvorverteilungsalgorithmus ist zuständig für das initiale Verteilen von Schlüsseln auf den Knoten, d.h. das Hinzufügen eines Knotens zum Netz. Der Algorithmus funktioniert inkrementell: Wenn ein neues Gerät zum Netz hinzugefügt werden soll, so muss auf sichere Art und Weise ein Schlüssel zu mindestens s Geräten, die sich bereits im Netz befinden ausgetauscht werden. Für den Schlüsselgraph bedeutet dies, dass beim Hinzufügen eines neuen

Knotens mindestens s Kanten zu bereits vorhandenen Knoten aus dem Schlüsselgraphen erstellt werden müssen. Falls der Graph insgesamt weniger als s Knoten enthält, muss man eine Kante zu jedem vorhandenen Knoten erstellen, d.h. für die ersten $s+1$ Knoten ist der Schlüsselgraph vollvermascht. Im Rahmen dieser Arbeit wird gezeigt, dass mit diesem Algorithmus immer ein s-verbundener Schlüsselgraph entsteht.

Wird ein Knoten später wieder aus dem Schlüsselgraph entfernt, kann die s-Verbundheit zerstört werden. Daher werden im Rahmen dieser Arbeit auch Algorithmen für das Entfernen von Knoten aus dem Schlüsselgraphen untersucht. Die Grundidee beim Entfernen ist eine Zurückführung auf den Schlüsselvorverteilungsalgorithmus: Man betrachtet, was gewesen wäre, wenn der entfernte Knoten niemals im Netz gewesen wäre und ändert daraufhin den Schlüsselgraphen dahingehend.

Schlüsselaustauschprotokoll

Es ist offensichtlich, dass man keine neuen Schlüssel aufbauen muss solange in dem Netz weniger als $(s+1)$ Knoten vorhanden sind – in diesem Fall ist der Schlüsselgraph vollvermascht. Daher betrachten wir für das Schlüsselaustauschprotokoll nur den Fall in dem es bereits mehr als $(s+1)$ Knoten im Netzwerk gibt.

Das vorgestellte Schlüsselaustauschprotokoll basiert auf einer Idee welche erstmalig in [Gon93] veröffentlicht wurde. Später wurde diese Idee in [CPS03] aufgenommen und für Sensornetze angepasst. Die Idee ist es, einen neuen Schlüssel in mehrere Teile aufzuteilen und die Teile auf unterschiedlichen Wegen zum Ziel zu schicken. Auf diese Weise muss der Angreifer alle Teile des Schlüssels ausspionieren um erfolgreich zu sein. Wenn zwei Geräte noch keinen gemeinsamen Schlüssel besitzen, aber sicher kommunizieren wollen so gehen sie wie folgt vor: Eines der Geräte erzeugt s zufällige Teilschlüssel[1]. Diese Teilschlüssel werden nun entlang von s knotendisjunkten Pfaden auf dem Schlüsselgraphen verschickt. Dabei werden sie auf jeder Teilstrecke mit dem entsprechendem Schlüssel verschlüsselt. Der endgültige Schlüssel wird von beiden Kommunikationspartnern durch die bitweise XOR-Verknüpfung aus allen Teilschlüsseln bestimmt. Ein ähnliches Verfahren wird auch in [CPS03] verwendet, allerdings wird hier keine feste Anzahl von knotendisjunkten Pfaden gefordert.

Der Angreifer kann den Schlüssel nicht rekonstruieren solange ihm noch ein Teilschlüssel fehlt. Daher ist dieses Verfahren absolut sicher, solange es dem Angreifer nicht möglich ist, mindestens ein Gerät auf jedem der s Pfade, d.h. mindestens s Geräte zu kompromittieren.

[1] Ein Teilschlüssel ist genau wie ein Schlüssel eine Bitsequenz einer bestimmten Länge

Fehlertolerantes Schlüsselaustauschverfahren

Das grundlegende Schlüsselaustauschverfahren ist nicht gegen Ausfälle von Geräten und gegen aktive Angreifer geschützt. Daher wurde das Schlüsselaustauschverfahren insofern erweitert, dass man Ausfälle von Geräten tolerieren kann. Des Weiteren wurden die Algorithmen so erweitert, dass es auch einem aktiven Angreifer nicht möglich ist, neue Schlüssel auszuspähen.

Fehlertoleranz wird erreicht indem ein gewisser Grad an Redundanz eingeführt wird. Um r Knotenausfälle zu tolerieren wird durch den Schlüsselvorverteilungsalgorithmus ein $(s+r)$-verbundener Graph aufgebaut. Dies bedeutet, dass es nun anstatt s knotendisjunkter Pfade mindestens $(s+r)$ knotendisjunkte Pfade zwischen beliebigen, nicht benachbarten Knoten gibt. Wenn ein Knoten erkennt, dass ein Schlüsselaustausch nicht erfolgreich war, so kann eine andere Menge von s knotendisjunkten Pfaden gewählt werden.

Im Rahmen dieser Arbeit wurden zusätzlich Algorithmen zur Wiederherstellung der ursprünglichen Verbundenheit des Schlüsselgraphen entwickelt. Dadurch ist es möglich insgesamt mehr als nur r Knotenausfälle zu tolerieren. Die Erweiterung des grundlegenden Algorithmus sind des Weiteren auch gegen aktive Angreifer geschützt.

Erweitertes Schlüsselaustauschverfahren

Das grundlegende Schlüsselaustauschverfahren basiert auf einer beliebigen Methode zum Auffinden von knotendisjunkten Pfaden in Graphen. Solch ein Algorithmus kann, je nach Algorithmus, eine zusätzliche Belastung für das Gerät darstellen. Insbesondere der Speicherverbrauch dieser Algorithmen ist in der Regel relativ hoch, wodurch sie in drahtlosen Sensornetzen mit ressourcenschwachen Geräten nur eingeschränkt einsetzbar sind. Daher wurde im Rahmen dieser Arbeit eine neuartige Möglichkeit entwickelt, diese knotendisjunkte Pfade zu finden bzw. zu verwenden. Diese Erweiterung des grundlegenden Schlüsselaustauschverfahrens ermöglicht den Einsatz des Schlüsselaustauschverfahrens auch auf höchst ressourcenschwachen Geräten.

Die grundlegende Idee bei der Erweiterung ist es den Schlüsselgraph zeitweise durch hinzufügen von neuen Kanten zu modifizieren. Im Detail, werden so lange neue Kanten in den Graphen eingefügt, bis zwischen den beiden Kommunikationspartnern s Pfade mit je genau einem Zwischenknoten existieren. Danach ist der eigentliche Schlüsselaufbau trivial. Man macht sich in dem erweiterten Verfahren zu nutze, dass man den Graph modifizieren kann – eine Gegebenheit, welche andere Methoden zur Suche nach knotendisjunkten Pfaden nicht vorraussetzen können. Im Rahmen dieser Arbeit wird gezeigt, dass mit dem erweiterten Schlüsselaustauschverfahren der Speicherverbrauch auf den einzelnen Geräten konstant gehalten werden kann. Das Nachrichtenaufkommen ist dabei linear in der Anzahl der Knoten.

Leistungsbewertung

In Rahmen dieser Arbeit wurden eine Reihe von Leistungsbewertungen durchgeführt. Zwei Kriterien werden im Folgenden genauer untersucht: der Speicherverbrauch auf den Geräten und die Robustheit des Verfahrens gegen den Angreifer.

Bei der Berechnung des Speicherverbrauchs wird als maßgeblicher Teil die Anzahl der zu speichernden Schlüssel pro Knoten betrachtet. Der Speicherverbrauch der einzelnen Geräte ist im Durchschnitt konstant und beträgt $2s$ Schlüssel. Dies kann man analytisch berechnen: Jeder neue Knoten bringt s neue Schlüssel in das Netz. Jeder Schlüssel muss auf zwei Knoten gespeichert werden, daher gibt es in einem Netz mit n Knoten genau $n \cdot 2s$ Schlüssel und somit im Durchschnitt $2s$ Schlüssel pro Knoten. Bei dem grundlegenden und dem fehlertoleranten Verfahren hängt der tatsächliche Speicherverbrauch zudem von dem verwendeten Pfadsuchalgorithmus ab. Da bei dem erweiterten Verfahren kein solcher Algorithmus zum Einsatz kommt ist in diesem Verfahren der Speicherverbrauch konstant. Ein konstanter Speicherverbrauch ist eine notwendige Bedingung damit das Verfahren auch für sehr große Netze eingesetzt werden kann.

Im Rahmen dieser Dissertation wurde die Robustheit des Verfahrens gegen Angreifer als die Wahrscheinlichkeit definiert, dass ein neu aufgebauter Schlüssel trotz c durch den Angreifer übernommener Geräte nicht kompromittiert ist. In Abbildung 1 wurde die Robustheit in Abhängigkeit von den kompromittierten Geräten c dargestellt. Das Netz besteht dabei aus $n = 100$ Knoten. Es wurden drei Sicherheitsstufen $s = 2, 5, 10$ verwendet und c bewegt sich im Interval $[0 \ldots n]$. Für das grundlegende bzw. das fehlertolerante Verfahren wird eine mittlere Pfadlänge $p_{avg} = 4$ angenommen.

Wie in der Grafik deutlich erkennbar, wächst die Robustheit des Verfahrens bei allen Verfahren mit steigendem s an. Das erweiterte Verfahren profitiert allerdings in stärkerem Ausmaß von einer größeren Sicherheitsstufe s als die anderen Verfahren.

Zusammenfassung

Mit den Schlüsselaustauschverfahren, die in dieser Arbeit vorgestellt werden, ist es möglich Sicherheit in drahtlosen Sensornetzen zu erreichen. Durch das Schlüsselaustauschverfahren stehen die notwendigen Schlüssel auf den einzelnen Geräten zur Verfügung, um die verschiedenen Sicherheitsziele wie Authentizität, Geheimhaltung, Integrität usw. zu erzielen. Die vorgestellten Verfahren verwenden theoretische Grundlagen aus der Graphentheorie um z.B. den erfolgreichen Schlüsselaustausch zu garantieren. Des Weiteren wird garantiert, dass der Angreifer keine neuen Schlüssel ausspähen kann solange er nicht mindestens s Geräte unter seine Kontrolle

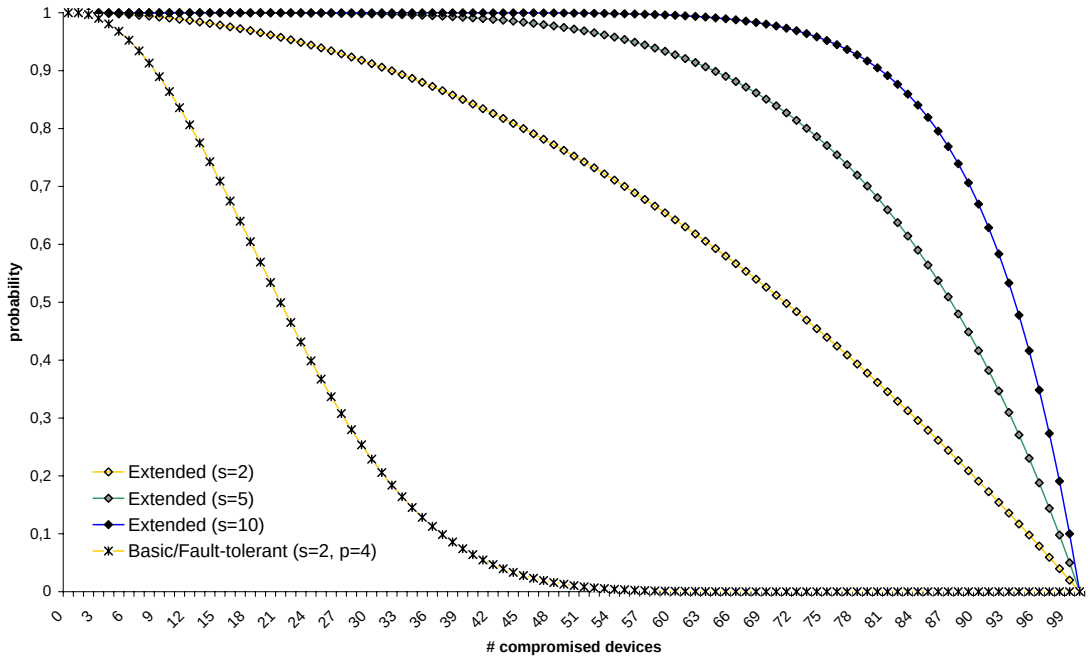


Abbildung 1: Robustheit gegen Angreifer für $s = \{2,5,10\}$, $n = 100$, $p_{avg} = 4$

gebracht hat. Diese Garantie ist ein wesentlicher Beitrag gegenüber bereits existierenden Ansätzen. Ferner wurden im Rahmen dieser Arbeit Algorithmen zum Hinzufügen und Entfernen von Knoten vorgestellt. Dadurch wird eine dynamische Größe des Netzes zur Laufzeit unterstützt.

Die Leistungsbewertung zeigt, dass sich die vorgestellten Verfahren in großen Netzen einsetzen lassen, wobei die Sicherheit schrittweise mit der Anzahl der vom Angreifer kontrollierten Geräte abnimmt. Wie in der Arbeit beschrieben, kann der Speicherverbrauch auf einen konstanten Wert beschränkt werden, wodurch nur wenige Ressourcen der Geräte für die Sicherheit benutzt werden. Die vorgestellten Schlüsselaustauschverfahren können daher in dezentralen drahtlosen Sensornetzen eingesetzt werden um einen Angreifen beispielsweise daran zu hindern, private Daten aus dem Gebäudemanagement auszulesen, böswillige Befehle an das Gebäudemanagement zu senden oder Falschmeldungen in das System einzuspeisen.

Durch die geringen Ressourcenanforderungen der vorgestellten Algorithmen, können sie überall dort eingesetzt werden, wo es darum geht Sicherheit auf kleinsten eingebetteten Geräten ohne zentrale Stelle bereit zu stellen.

Contents

List of Figures

List of Algorithms

List of Procedures

1

Introduction

In recent years wireless sensor networks have emerged in a variety of application domains. They are used for wildfire detection [DS05], wildlife surveillance [MCP+02] and in agricultural production [BBB04]. In addition, sensor and actuator networks are used in Pervasive Computing [Wei91] scenarios, e.g. home automation. Here a private home is equipped with a multitude of sensors and actuators to enhance the lifestyle of individuals. For instance, the heating is turned on automatically when the owner of the house is about to arrive home; the light is switched on in rooms where motion is detected, or the door is automatically opened when an authorized person approaches the door.

To realize such sensor networks, different design aspects must be taken into consideration. As an example, sensor devices must be designed to be cost effective and small enough to be embedded in the environment. In addition, wireless communication technologies must be developed to provide energy efficient data transmission between the sensor devices while allowing long device life times.

Security is a crucial factor for the success of such wireless sensor networks. Inaccurate or manipulated readings e.g. through vandalism could pose a major problem in monitoring sensor networks. As an example, if enough false data is injected into the network, false wildfire alarms could lead to panic and ultimately reduce the system's acceptance. Furthermore, in the home automation scenario, many new ways to invade an individual's personal life are introduced. Hackers could query sensor data to monitor people's behavior in their home environments, largely violating their privacy. A thief could check if somebody is at home before breaking into a house. A modern vandal might attempt to shut down the heating in the middle of the winter. Hence, securing wireless sensor and actuator networks is crucial for the general acceptance of this technology. Sensor data must be encrypted to ensure that only authorized persons are able

to read it. Access to sensors and actuators must be restricted to circumvent misusing them. Manipulated data must be prevented from entering the network.

The provision of suitable security solutions is not easy. Due to the severe resource restrictions experienced in sensor networks, e.g. with respect to computation, memory, bandwidth and energy, existing solutions for conventional desktop computers and wired networks cannot be applied. Instead, specialized solutions must be developed, which are specifically tailored towards sensor networks. In this dissertation we present a solution for securing wireless sensor and actuator networks. The resulting area of research is depicted in Figure 1.1.

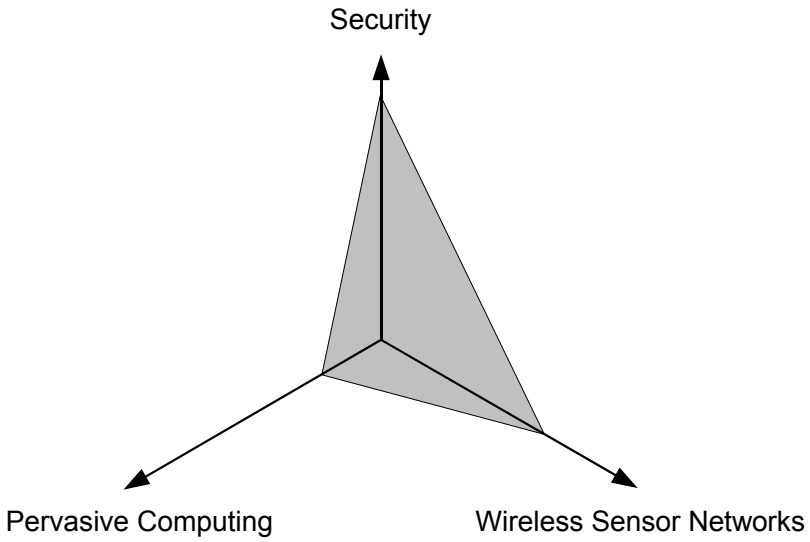


Figure 1.1: Area of research

More specifically, we address the problem of key distribution in such networks, which is fundamental for any cryptography-based security system. To do so, we develop three different key distribution schemes and analyze their behavior in different scenarios.

1.1 Security in Wireless Sensor Networks

To provide a secure wireless sensor network, a multitude of security techniques must be integrated into the network. The foundation for most of these techniques is encryption. Encryption can be used to provide confidential data exchange, user authentication, data integrity, etc. In modern cryptographic systems, data encryption relies on the usage of keys. Keys are arbitrary bit sequences of suitable lengths, and are used to encrypt and decrypt data. Whoever owns a certain key can decrypt the corresponding data items. Without the key, decryption is impossible.

In general, cryptography can be classified in *symmetric* and *asymmetric* algorithms [Sch95]. Conceptionally, symmetric algorithms – also referred to as secret key algorithms – use the same key for encryption and decryption. Hence, to exchange data securely between two communication partners, the key must be known to both partners and is often referred to as the shared secret. Before sending a message, the sender encrypts it using the shared key. Upon reception, the receiver decrypts it with the same key. Symmetric algorithms are based on transposition and substitution and can be implemented very efficiently in soft- and hardware. Well known examples for such algorithms are DES [DES77] and AES [DR02, AES01].

In contrast to symmetric algorithms the asymmetric ones – also referred to as public-key algorithms – use different keys to encrypt and decrypt data, the *public* and the *private* key. To enable secure communication, each communication partner generates his own private and public key. While the public key is published, and therefore known to any potential communication partner, the private key remains secret. Messages which are encrypted with a certain public key can only be decrypted with the corresponding private key. Hence, to send an encrypted message to a given receiver the message is encrypted using the receiver's public key. The receiver can then decrypt the message using his private key. Asymmetric cryptography involves modular arithmetic on very large numbers (i.e. numbers with several hundred decimal places[1]), resulting in much more computational overhead than symmetric cryptography. Therefore, asymmetric cryptographic operations are much slower to compute than symmetric ones. Well known examples for asymmetric algorithms are RSA [RSA78] and ECC [Mil86].

The devices used in wireless sensor networks are typically highly resource-constrained [PSW+01]. To save the devices' scarce resources symmetric cryptography is much better suited to be used for such devices than asymmetric cryptography. Hence, existing security systems for wireless sensor networks usually rely on symmetric cryptography [PSW+01, KSW04, CP05].

However, to enable symmetric cryptography, the keys must be securely distributed between the communication partners without others overhearing them. To do so, a suitable key distribution scheme for wireless sensor networks is needed. In general, key distribution schemes can be classified using two dimensions: first, the cryptographic algorithms used for the key exchange and second, the system organization. Regarding the cryptographic algorithms used we can distinguish between solutions using symmetric cryptography and solutions using asymmetric cryptography. Regarding the system organization, centralized and decentralized solutions are possible. A key distribution scheme may either use some kind of centralized authority, like e.g. a base station or an authentication server, or it may operate in a fully decentralized way. Hence we can derive four different classes as depicted in Figure 1.2.

The topic of this dissertation is located in the lower right section of Figure 1.2, i.e. key distribution schemes using only symmetric cryptography in a purely decentralized setting. In the

[1] As a comparison 10^{84} corresponds to the volume of the entire universe in cm^3 [Sch95]

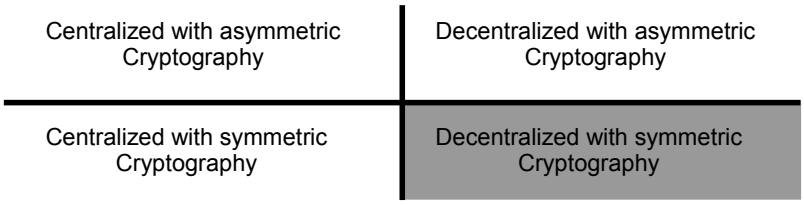


Figure 1.2: Classification of key distribution schemes

following we discuss all four classes in order to motivate this decision.

1.2 Key Distribution Schemes

A straight forward approach to exchange keys is to provide a centralized authority for the key exchange, i.e. a key distribution center (KDC) [PK79]. At system startup time, each device in the network shares a unique symmetric key with the KDC. To communicate with another member of the network, a device contacts the KDC securely and requests a new key between itself and the other device. The KDC creates a new key and sends it securely to both communication partners. Using this new key the devices can exchange encrypted data with each other. Well known examples[2] for this approach are [KN93, NS78].

However, using a KDC has a number of drawbacks. First, the KDC is a single point of trust. If it is subverted by an attacker, the whole system falls and no more security can be guaranteed. Second, the KDC is a single point of failure. If it becomes unavailable, e.g. because of hardware failure, no more keys can be established.

To tolerate the failure of a KDC decentralized approaches are needed. Such approaches do not rely on a central authority. Instead, multiple devices cooperatively establish and distribute new keys. Decentralized approaches can be based on asymmetric cryptography (e.g. [ZH99, HBC01]) or symmetric cryptography (e.g. [CPS03, ZXSJ03, CP05]). Symmetric key establishment is based on a scheme first proposed by [CPS03] in 2003 and inspired by [Gon93]. The basic idea behind Gong's approach is to split the secret among a number of devices, whereas a single one does not have the entire knowledge about the secret. Chan et al. adapted this approach for key establishment in wireless sensor networks by splitting the key to be established into parts, so-called *key shares*. To compromise the key, an attacker must recover all parts. Two devices that do not share a key yet establish a new one as follows: one of the devices randomly generates a number of key shares, and sends them over device-disjoint paths – i.e. paths that do not share common devices – to the destination device. On each hop of a path, the key share is encrypted using the existing shared key for this link. The final key is calculated

[2] The presented examples also provide authentication

as the bitwise exclusive-or operation of all key shares. Clearly, these approaches rely on the existence of a sufficient number of device-disjoint paths. Otherwise, key establishment is not possible. However, existing approaches cannot guarantee the existence of such paths and thus the operation of the key establishment [CPS03, ZXSJ03].

In order to guarantee the establishment of symmetric keys between arbitrary devices in a wireless sensor network, this thesis presents a novel approach based on symmetric cryptography. This approach tolerates the failure of a KDC and thus operates in a decentralized way. In addition, it ensures the existence of multiple device-disjoint paths in the network to allow the exchange of key shares in any given situation.

1.3 Contributions

In this dissertation we present a suite of algorithms to exchange symmetric keys in wireless sensor networks without the need for a central authority. These algorithms are suitable for resource-constrained devices like sensors and avoid a single point of trust. Unique keys are exchanged between device-pairs providing authenticity. Even if a device is subverted by an attacker, the key exchange for the remainder of the network remains functional.

More specifically, the main contributions of this work are as follows:

- We derive and analyze the requirements for a key distribution scheme for wireless sensor networks.
- We provide a mathematical model for such schemes based on graph theory. This allows us to adapt graph theory definitions and theorems, namely the concept of the *s*-connectivity of a graph. Furthermore, with the Menger theorem [Men27] we can formally prove the existence of a certain number of device-disjoint paths in our network, which is a major foundation for our approach.
- We develop a novel key distribution scheme using the previously developed mathematical model which includes algorithms for adding devices to the network, removing them, and establishing new keys at runtime. These algorithms work on the mathematical foundations provided by the graph theory. The key establishment algorithm is based on an adequate path search algorithm based on graph theory.
- We extend our key distribution scheme to offer resilience against multiple concurrent device failures and more powerful attackers, i.e. active attackers as defined in Section 2.1.
- We provide a third, extended key distribution scheme, which works more efficiently. This is achieved by providing a novel approach for detecting device-disjoint paths in a wireless sensor network.

We complement these contributions with a detailed performance evaluation of our algorithms.

1.4 Structure of this Dissertation

This dissertation is organized as follows: In Chapter 2 we present our system model, the requirements for our key exchange scheme, and the design rationale for our approach. In the following chapters we then present our our approach, i.e. the key distribution schemes. To present our approach more comprehensibly we divide its description in three parts.

In the first part, we consider *passive attackers* only. We call this our basic key establishment scheme since the two other parts are based on it. This approach is described in Chapter 3.

In the second part we extend the basic approach to a fault tolerant approach in order to cope also with device failures and a stronger attacker type – the *active attacker*. This approach is described in Chapter 4.

The basic approach relies on path search algorithms. Since such a path search might not be efficient enough in order to be usable on highly resource-constrained devices we extend our approach in the third and final part in order to cope also with such devices. This approach is described in Chapter 5.

After presenting our three key exchange schemes, we evaluate them in Chapter 6. In Chapter 7 we present other existing approaches for key establishment and compare them to our approach. Finally, we conclude this dissertation in Chapter 8 with a summary and an outlook on future work.

2

Foundations

In this chapter we provide the foundations to develop our novel approach for establishing pairwise keys between arbitrary devices in a wireless network. Our approach is well suited for highly resource-constrained devices and works in a fully decentralized way, i.e. there is no single point of failure.

In Section 2.1 we formalize the system model used throughout the rest of this work. The requirements regarding our approach are discussed in Section 2.2. We then present in Section 2.3 the design rationale of our approach. In Section 2.4 we present some definition and basic data structures which are needed for the description of our approach.

2.1 System Model

Our system model comprises four parts. We start with our network model by describing the characteristics of our network. The properties of our devices are given next in the description of our device model. After that the assumed failure and attacker models are presented.

Network Model

For the purpose of this work we assume that the network consists of a multitude of independent devices that communicate over a wireless channel. The channel itself is insecure, i.e. anyone can listen and send to the channel.

Due to the wireless nature the channel is error-prone. We therefore assume the existence of a transport layer that recovers from packet loss. However, we analyze the effects of different error-rates under a very simply error recovery transport layer in Chapter 6.

We further assume a non-partitioned network so that communication between any two devices is always possible (direct or indirect via ad-hoc routing). One possibility to achieve a non-partitioned network is to deploy the devices with a high density in a certain area – in ideal case they would all be in the communication range of each other, hence no partitioning can happen. Also each device has its network wide unique device ID, through which it can be uniquely addressed.

The number of devices in our network is not predetermined or constrained in any way, and may change due to the introduction of new devices, device deactivation or failure.

Device Model

Our devices are independent with their own memory and processor. We assume that they have some sort of energy source e.g. batteries. Due to the huge amount of such devices – and their usage as sensor- or actuator-devices in daily life items – they must be inexpensive and have therefore very limited resources. Typical devices are for instance based on eight bit micro-controller with only a few hundred bytes of RAM and a few kilobytes of program memory (e.g. flash memory).

We further assume that such devices are not in any way tamper resistant. This assumption is based on the fact that tamper resistance is very hard to achieve [AK96] and even harder if the used devices must be very cheap. This means, that having physical access to such a device, it is highly probable that all data stored on it can be recovered from its memory. This means that any secrets that are stored on such a device, e.g. cryptographic keys for communication can be retrieved by a person who is able to gain physical access to the device.

Failure Model

In order to formalize our failure model we introduce the notion of a *process*. On each device d_i our protocol is executed by a process p_i (Figure 2.1). The process p_i is responsible for sending and receiving messages on the network. For the purpose of this work we assume that p_i for all i shows a *fail-stop* behavior in case of an failure. For instance, a device which batteries died, exhibits a fail-stop behavior.

Attacker Model

We formalize our adversary as a process a_i running on a subverted device d_i. The attacker model is presented in Figure 2.2. In this model process a_i only communicates with process p_i, which executes the communication protocol. Therefore, a_i never communicates directly with

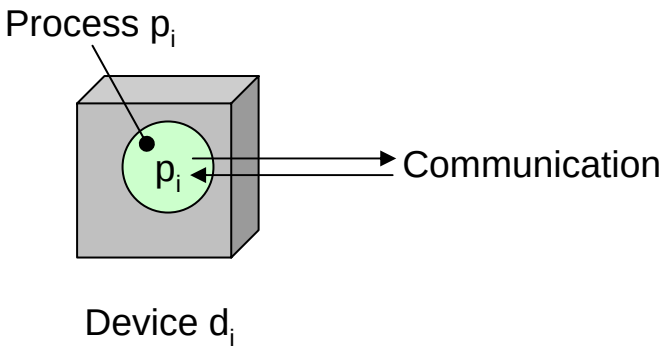


Figure 2.1: Failure Model

any other process p_j on a device d_j for all j. However we assume, that all processes a_i may communicate with each other in order to reach their goal of compromising secrets, i.e. newly established keys. For this communication any means are possible. For instance an attacker may place several subverted devices in our network, where the subverted devices additionally communicated over a wired high speed network.

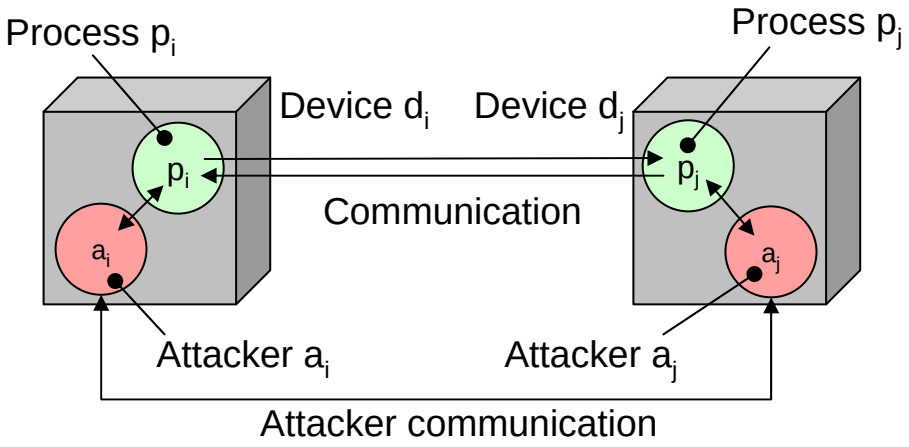


Figure 2.2: Attacker Model

In security applications an attacker can be classified by two main categories, i.e. his objective or goal and his ability. With his objective we describe the intentions of an attacker, e.g. gathering secret knowledge. The ability of an attacker describes what an attacker is able to do in order to reach his goal. With this model in our mind, we define the following types of attackers by presenting for each attacker type his objective and ability:

1. The *passive eavesdropping attacker (**Eve**)*. The objective of this attacker is to eavesdrop on communication between devices. This attacker is only interested in learning about secrets of other devices, e.g. newly established keys. The ability of this attacker can be

modeled as a process a_i on a subverted device d_i, where a_i has reading access to the process p_i. This means, that a_i can learn about any messages sent or received by p_i and even all memory contents of p_i. However, a_i can never trigger p_i to send any malicious messages.

2. The *active intruding attacker (**Trudy**)*. The objective of this attacker is also – just like the eavesdropping attacker – to learn about new secrets of other devices, i.e. newly established keys. However, unlike the eavesdropping attacker, this attacker may trigger the injection of messages into the network in order to reach his/her goal. Hence, the ability of this attacker can be modeled as a process a_i on a subverted device d_i, where a_i has full, i.e. read/write access to the process p_i. He therefore can additionally to the passive eavesdropping attacker also trigger p_i to send messages in order to gain access to secrets while having the same objective. This attacker subsumes the passive eavesdropping attacker class.

3. The *active malicious attacker (**Mallory**)*. The objective of this attacker is to hinder correct execution of the networks purpose while having the same abilities as Trudy. Hence, the ability of this attacker can be modeled exactly as Trudy. However, due to the different goal of this attacker he becomes a very powerful *denial-of-service* attacker. This attacker subsumes both attacker classes from above.

In Figure 2.3 we summarize the abilities and the goals of our three attacker types. Note that Eve and Trudy have a common goal while Trudy and Mallory have the same abilities.

	Eve	Trudy	Mallory
Goal	Capture new keys	Capture new keys	Denial-of-Service
Ability	Listen	Listen/Send	Listen/Send

Figure 2.3: Attacker Classification

For the purpose of this work we assume only Eve- and Trudy-type attackers with their common goal of learning about new secrets and their different abilities. Mallory-type attackers are beyond the scope of this work and subject of future research. Note that Mallory-type attackers are

more powerful only due to the different goal, i.e. denial-of-service which is a general, mostly unsolved, problem not only for wireless sensor networks.

In this work we do not consider attacks on the physical layer like jamming the frequency. Such attacks are also beyond the scope of this work and can not be counteracted on the transport layer.

2.2 Requirements

The system model described before leads to a number of requirements that must be fulfilled by a secure key distribution scheme for wireless sensor and actuator networks. In the following, we describe these requirements.

Decentralized Operation

The first crucial requirement for a key distribution scheme in a network of sensors and/or actuators is that it remains operational even if some devices fail or get subverted by an attacker. Thus, the network should not contain any specialized devices, which need to act as a central server, e.g. key-server. Any such central server would clearly be a single point of failure and further a designated target for any possible attacker. Therefore, designing a secure and robust key distribution scheme requires a fully decentralized algorithm.

Symmetric Cryptography

The hardware used for sensors and/or actuators is usually highly resource-constrained because the devices should be cheap and very small. This implies a limitation on the useable cryptographic algorithms since the devices should have enough resources left despite cryptography to perform their actual functionality. Asymmetric cryptography uses much more resources than symmetric cryptography [Sch95, CKM00, CP05]. As an example, RSA with a key length of 1024 bit needs approximately 4737s to decrypt a message on a PIC18 processor used for the SmartIts [Sma] sensor platform [Gir05]. ECC – which is specifically intended for resource-restricted devices – with 192 bit key length needs about 13.9s for the same operation. For a complete key exchange using ECC eight sequential such operations are needed, leading to a total of 111.2s. In all these cases, the processors were fully occupied to perform the cryptographic operations and did not execute any other application code. As a comparison, the symmetric DES algorithm needs approximately 0.018s to decrypt a 128 byte message on a PIC16 processor [Smi03], about 263166 times faster than RSA and 769 times faster than ECC. To obtain these values, we conducted a number of experiments for different hardware platforms.

For more results please refer to [Wur03, Gir05]. Following our results, to deal with the resource limitations of the used devices we require that our key distribution scheme uses only symmetric cryptography. One might argue, that with the fast going development on computer hardware these limitations will vanish within the next few years. Karlof and Wagner made an interesting observations with respect to this argument [KW03]: Sensors and actuators are more likely to ride Moore's law *downward*. Instead of doubling the available resources on these devices, they will more likely become even smaller and cheaper.

Graceful Degradation

Closely related to the fact that the used devices being cheap and small is that we cannot expect them to be tamper resistant. Assuming the presence of an attacker together with the fact of the devices not being tamper resistant (see also Section 2.1), the proposed key distribution scheme must cope with some subverted devices. Thus, another requirement for such an key distribution system is to stay operational even if a certain number of devices is subverted by the attacker. Clearly, the subverted devices are lost to the attacker - however we require that for any other devices which are not compromised, the secrecy of newly established keys remains guaranteed. In the literature this is often called *graceful degradation* of the network [KW03].

Dynamic Network Size and Scalability

In many scenarios the size of the network cannot be predetermined. If, for instance, a sensor and actuator network is used for home automation, the user wants to add devices at a later point in time or remove older devices from the network. Thus, we require that our key distribution scheme supports dynamic network sizes. Furthermore, it should be easy to introduce devices in an already existing network and to remove them from the network. To be useable for large networks with many devices, both the key distribution scheme itself as well as the introduction and removal of devices should be scalable.

Cope with Device Failures

Devices in sensor and actuator networks may fail, e.g. due to depleted batteries or physical stress placed on the devices. A key distribution scheme for such networks has to cope with failing devices. If a certain device fails, the operation of the rest of the network should not be hindered in any way.

Guaranteed Key Establishment

Whenever two devices in a sensor and actuator network need to communicate securely with each other, they need to be able to establish a secret key. Thus the last – but not less important – requirement for the key distribution scheme is to provide a guarantee that at any time any two devices in the network can establish a new secret key securely. Situations in which a given device pair may not be able to do so are unacceptable. We call this guaranteed key establishment or guaranteed operation of the key distribution scheme. Most existing approaches are probabilistic (see also Section 2.3), i.e. they cannot guarantee that between all pairs of devices a key can be established successfully.

2.3 Design Rationale

After formalizing our system model and discussing the requirements in the last sections we are now ready to develop our new key distribution scheme. As stated in Section 2.2, due to the device's resource limitations only symmetric cryptographic algorithms can be used. Therefore, whenever two devices of such a network need to communicate securely with each other they must share a common symmetric key.

The simplest approach is to share a common (global) key between all devices in the network. The advantage of this approach is the simplicity of its implementation. However, when a single device gets subverted by the attacker, the global key is known to the attacker and thus the whole network is compromised. Clearly, this approach does not comply with our requirements since it does not cope with a single subverted device.

In order to cope with compromised devices we need to reduce the number of devices that use the same key. The minimum number of devices which use a certain key is two, which is called *pair-wise* keys. Clearly, pair-wise keys provides the highest resilience against subverted devices (due to the *unique* common secret). Therefore the goal of most current key distribution schemes is to provide pair-wise keys for any device pair of the network. However, due to the device's resource limitations it is usually not possible to store all keys on each device. Instead, only a subset of all keys can be stored. This leads to two questions: first, which subset to store on which device, and second, how to reconstruct the remaining keys when needed. Therefore a typical key distribution scheme comprises of two parts: the *key pre-distribution* and the *key establishment*. The key pre-distribution is responsible for selecting and storing the key subsets on the devices. This is usually assumed to be done out-of-band in a secure way. As an example, a manufacturer of a sensor network can place key pairs on the sensor devices at production time in his secure environment. Using the key subsets stored by the key pre-distribution, the key establishment creates the remaining keys whenever they are needed. This is usually performed

after the devices are deployed in the field. Therefore, in contrast to the key pre-distribution, the key establishment must tolerate the presence of an attacker.

We can classify the *key pre-distribution* protocols depending on their algorithm used for the choice of which devices will share a key. Most protocols choose the devices which share a unique key randomly, hence they are called *random key pre-distribution* protocols. Due to the random choice of the pre-distributed set of keys, these protocols cannot guarantee that a key establishment protocol will be able to establish a new key in all cases. Hence, random key pre-distribution protocols are of a probabilistic nature and therefore cannot fulfill our requirement for guaranteed key establishment from last section. The basic foundation for providing guaranteed key establishment by the key establishment protocol is to choose the pre-distributed set of keys in a non-random way, i.e. after a well defined algorithm. Such approaches are called *controlled key pre-distribution* protocols. Hence, due to our requirements we develop in this work a controlled key pre-distribution protocol, where the pre-distributed key set is chosen by a non-random algorithm leading to a non-probabilistic key distribution scheme.

As already stated, whenever two devices which do not share a common key need to communicate securely, they need to establish a new one. The most common way to do so is to find a multi-hop path between both devices, such that each device on the path shares a key with its predecessor and successor. This is commonly called a secure path [DDH$^+$04, LLZ05]. For establishing a new key, the key is then send over this secure path. Since all intermediary devices on the path can read the new key, a single subverted device on the path can compromise it. Therefore in our approach we fragment the key into key shares and send each share over a different secure path. To reconstruct the newly established key, an attacker needs to compromise at least one device on each secure path in order to intercept all key shares. The key pre-distribution proposed in this dissertation guarantees that these distinct secure paths exist. By adapting the number of key shares and thus the number of secure paths we can control the resilience of the network against the attacker, the so-called *security level*.

2.4 Definitions

In this section we give some definitions that are needed to explain our approach. We start by defining the already mentioned security level. After that we introduce a formal representation of our network with the aid of three graphs, the key graph, the communication graph, and the physical neighborhood graph.

Security Level

With our approach we introduce the *security level s* of the network. This is a parameter which can be freely chosen by the network administrator in order to control the resilience of the network against the attacker. We define s as follows: As long as the attacker has compromised fewer than s devices, we can guarantee that the attacker cannot achieve his goals. In the case of Eve or Trudy, this means that any new keys are secure i.e. the attacker cannot discover them. If the attacker compromises at least s devices in the network, we cannot guarantee anymore the secrecy of new keys. Note that even if the attacker compromises s or more devices, he will not in all cases be able to discover all new keys. This is due to the graceful degradation of the network security. This is further analyzed in Section 6.3.1.

Redundancy Level

We introduce the *redundancy level r* of the network. This parameter can be freely chosen by the network administrator in order to control the resilience of the network against device failures. We define r as follows: As long as there are less than r devices failures in the network, we can guarantee that two arbitrary devices can establish a new key. If more than r devices failed then new key establishments cannot be guaranteed anymore.

Key Graph

We introduce the notion of a *key graph* as an undirected graph $G_K = (V, E_K)$, where V is the set of devices in the network, and E_K represents the set of shared keys between devices where $\{v_1, v_2\} \in E_K$ iff the nodes v_1 and v_2 share a symmetric key due to the key pre-distribution. We further will use the term *device* to indicate the physical device and the term *node* to indicate the representation of that device in the graph.

Communication Graph

We define the *communication graph* as an undirected graph $G_C = (V, E_C)$, where V is the set of devices in the network and E_C represents all communication links between the devices. That is $\{v_1, v_2\} \in E_C$ iff the devices represented by the nodes v_1 and v_2 are able to communicate with each other. Note that this communication might be direct or indirect via some routing layer, i.e. the communication graph represents the network connectivity. Furthermore we assume an undirected graph, i.e. if a device A can communicate with a device B, B can also communicate with device A. This can be achieved with an appropriate routing mechanism – see also next paragraph.

Neighborhood Graph

The neighborhood graph is also defined as an undirected graph $G_N = (V, E_N)$, where V is the set of all devices in the network and E_N represents the neighborhood relationship between the devices. This means, that $\{v_1, v_2\} \in E_N$ iff the devices represented by the nodes v_1 and v_2 are in physical communication range with each other, i.e. they can communicate directly. For the sake of simplicity, we assume G_N as an undirected graph, i.e. the communication ranges of the devices are symmetrical.

Notation

We use throughout this document big capital letters like $A, B, C, \ldots$ to represent device IDs. The corresponding nodes in one of our graphs are denoted by $v_A, v_B, v_C, \ldots$. Additionally small letter as the index of a node, e.g. $v_i, v_j, \ldots$ are used to generally refer to nodes within a graph. Small letters, when not used in indices, are used in order to denote system parameters like for instance the security level s or the redundancy level r.

2.5 Data Structures and Basic Procedures

In this section we introduce some data structures and primitives which we will use throughout the rest of this document.

2.5.1 Data Structures

In the description of our algorithms and associated procedures, the following data structures are used:

MyDeviceID: As mentioned in Section 2.1, each device has its own unique device ID. *MyDeviceID* is a variable, which is globally available on each device and contains the unique device ID of the device.

KeyStore: The (main) *KeyStore* contains all keys that a device shares with other devices due to the key pre-distribution. The main key store is organized as a list of 2-tuples $kse = (k, ID)$, where k is the shared key and ID is the device ID of the device corresponding to the key. Furthermore, we define the following methods on the *KeyStore*: *AddKey*, *GetKey*, *DeleteKey*, *ReplaceKey*, *KeyExist*, *GetIDRange*, and *GetCount*.

***AddKey*:** This method adds a new key to the end of the list. It takes two arguments namely *ID* and *key*, creates a tuple (key, ID) and adds it to the *KeyStore*.

***GetKey*:** With this method a key is retrieved from the *KeyStore*. It takes as an argument the *ID* of the device corresponding to the key and returns the key.

***DeleteKey*:** This method removes a *kse*-tuple from the *KeyStore*. To identify the tuple that should be removed, the method takes one argument, *ID*, denoting the ID of the device that this key is shared with. Since the *KeyStore* is an ordered list, all remaining tuples are shifted in order to fill the now empty place.

***ReplaceKey*:** This method exchanges one *kse*-tuple from the *KeyStore* with another. It uses three arguments, namely ID_{old}, *ID* and *key*. If there exist a tuple (ID_a, key_a) in the *KeyStore*, calling ReplaceKey(ID_a,ID_b,key_b) would result that in the location where the tuple (ID_a, key_a) was stored, after the call would be (ID_b, key_b).

***KeyExists*:** This method returns true if a *key* exists in the *KeyStore* for a given *ID*.

***GetIDRange*:** This method takes the two indexes – *start* and *end* – as arguments. It returns the list of device *ID*s being stored in the KeyStore from index *start* to *end* (inclusive).

***GetCount*:** This method returns the total number of keys stored in the *KeyStore*.

Furthermore, we define some additional methods on the key store, which are just simplifications for often used functions, i.e. they do not introduce any new functionality to the key store.

***GetAllIDs*:** This method is an alias for *KeyStore.GetIDRange(0,KeyStore.GetCount*-1*)*, i.e. it returns a list of all IDs to which a key is stored in the key store.

***GetParentsSet*:** This method is an alias for *KeyStore.GetIDRange(0,s* -1*)*, i.e. it returns the first *s* IDs that have been added to the key store (using *AddKey*).

***GetChildrenSet*:** This method is an alias for *KeyStore.GetIDRange(s,KeyStoreGetCount*-1*)*.

EstablishmentKeyStore: Keys which are established with the key establishment protocol are stored in the *EstablishmentKeyStore* or secondary key store. The secondary key store is organized as a list of 3-tuples $ekse = (k, ID, counter)$, where k is the shared key and *ID* is the device ID of the device corresponding to the key. As will be seen, any established key is calculated by XORing so-called *key-shares*. Thus, the third element of the tuple, $counter$, contains the number of key-shares which have been applied to this key. A key in the *EstablishmentKeyStore* is only valid if $counter \geq s$, i.e. the key is composed out of at least s shares. Furthermore we define the following methods on the *KeyStore*: *AddKey*, *GetKey*, *DeleteKey*, *UpdateKey* and *KeyExist*.

***AddKey*:** This method adds a new key to the end of the list. It takes two arguments namely *ID* and *keyshare*, creates a tuple $(keyshare, ID, 1)$ and adds it to the *EstablishmentKeyStore*.

***GetKey*:** With this method a key may be retrieved from the *EstablishmentKeyStore*. It takes as an argument the ID of the device corresponding to the key and returns the key.

***DeleteKey*:** This method removes a *ekse*-tuple from the *EstablishmentKeyStore*. To identify the tuple that should be removed, the method takes one argument, *ID*, denoting the ID of the device that this key is shared with.

***UpdateKey*:** This method is used to update a key. It takes two arguments namely *ID* and *keyshare*. If k_{ID} is the key which is already stored in the *EstablishmentKeyStore*, then *UpdateKey(ID, keyshare)* performs the following operation: $k_{ID} := k_{ID} \oplus keyshare$. Additionally, the *counter* corresponding to this *ID* is incremented by one, i.e. $counter_{ID} := counter_{ID} + 1$. If this method is called with an *ID* which does not exist in the *EstablishmentKeyStore*, the call is passed over to *AddKey*.

***ValidKeyExists*:** This method returns true if a valid *key* exists for a given *ID*. A key is valid if it has a *counter* value of *s*.

2.5.2 Basic Communication Procedures: send and receive

The basic procedures for the communication task are `send`, `broadcast`, and `receive`. When the device needs to transmit a (unicast) message to some other device over the air procedure `send` is called. The basic procedure `send` takes three arguments: *Destination*, *MessageType*, and *Payload*. The argument *Destination* takes the device ID of the receiving device. *MessageType* identifies the type of the message. The argument *Payload* contains the data to transmit to the other device.

When a device needs to send a message to all other devices in the network, procedure `broadcast` is called. It takes two arguments: *MessageType* and *Payload*. While *MessageType* identifies the type of the message, *Payload* contains the transmitted data.

Similar to procedure `send`, whenever a new message is received over the air, procedure `receive` is executed. It has three arguments: *Source*, *MessageType* and *Payload*. *Source* contains the device ID of the sending device. *MessageType* is the type of the message and *Payload* contains the payload which has been send by the source device.

As already mentioned in our network model (Section 2.1) we assume the existence of a protocol which recovers from communication failures, such as garbled, lost and duplicated messages. Hence, the two procedures `send` and `receive` provide a reliable datagram service. The procedure `broadcast` however, provides an unreliable datagram service.

2.5.3 Basic Cryptographic Procedures: encrypt and decrypt

The basic cryptographic procedures `encrypt` and `decrypt` use internally some symmetric cryptographic algorithm, e.g. AES or 3DES. Both procedures take as the first argument the used key. The length of the key should be chosen in order to suffice the actual resilience. The procedure `encrypt` takes as the second argument the message in cleartext and returns the encrypted message. The procedure `decrypt` takes as the second argument the encrypted message and returns the message in cleartext. We assume, that the procedure `decrypt` is able to detect whether the provided key is wrong. For example, this can be done by requiring a certain message format.

2.5.4 Secure Communication Procedures: SecureSend and SecureReceive

In order to provide secure communication between two devices we introduce the procedures `SecureSend` and `SecureReceive`. These two procedures use the basic communication procedures `send` and `receive` together with the *KeyStore* and the basic cryptographic procedures `encrypt` and `decrypt`. The pseudo-code for these procedures is given in Procedure 2.1 and 2.2 respectively.

```
1:  procedure SecureSend(Destination, MessageType, Payload);
2:  if KeyStore.KeyExist(Destination) then
3:      Key_Destination := KeyStore.GetKey(Destination);
4:  else if EstablishmentKeyStore.KeyExist(Destination) then
5:      Key_Destination := KeyStore.GetKey(Destination);
6:  else
7:      return false;
8:  end if
9:  EncryptedPayload := encrypt(Key_Destination, Payload);
10: send(Destination, MessageType, EncryptedPayload);
11: return true;
```

Procedure 2.1: Secure sending

When sending a new message to another device in the network, the procedure `SecureSend` is called with the payload *Payload* in cleartext. First, the device checks if a key is available for the intended *Destination*. In case a key is found in the *KeyStore* the message is encrypted using this key and then send using the basic communication procedure `send`. If there is no key available, the procedure returns **false** in order to signalize the failure of transmission.

Receiving a secure message is similar to sending a secure message. When receiving a new message from the network, the procedure `receive` is called. The device first checks if there is

```
1: receive(Source, MessageType, EncryptedPayload);
2: if KeyStore.ValidKeyExist(Source) then
3:     Key_Source := KeyStore.GetKey(Source);
4: else if EstablishmentKeyStore.KeyExist(Source) then
5:     Key_Source := EstablishmentKeyStore.GetKey(Source);
6: else
7:     // Discard message
8:     return ;
9: end if
10: Payload := decrypt(Key_Source, EncryptedPayload);
11: SecureReceive(Source, MessageType, Payload);
```

Procedure 2.2: Secure receiving

a key available in the *KeyStore* for the corresponding *Source*. In case there is no key found, the message is discarded. If a key can be found than the message is decrypted and the decrypted message is passed on to the procedure `SecureReceive`, which handles the message.

2.5.5 Message Dispatching

Messages are distinguished by a type field. Whenever a secure message is received, the type field of this message must be examined and then an appropriate message handler must be called. The actual procedure is given in Procedure 2.3. In order to simplify the procedures in the rest of this document, we state the following: a procedure of the form `onReceive<MessageType>` is the procedure which is called upon receiving a message with the type "MessageType".

```
1: procedure SecureReceive(Source, MessageType, Payload);
2: if MessageType == X then
3:     onReceiveX(Source, MessageType, Payload);
4: else if MessageType == Y then
5:     onReceiveY(Source, MessageType, Payload);
6: else if MessageType == <MessageType> then
7:     onReceive<MessageType>(Source, Payload)
8: else
9:     Unknown type, discard message
10: end if
```

Procedure 2.3: Dispatching received messages

2.6 Summary

In this chapter we formalized our system model and the requirements. Furthermore we motivated our design rationale and provided some definitions, data structures, and basic procedures used throughout the rest of the document. We describe our approach for establishing pairwise keys between arbitrary devices in a wireless network in the next three chapters.

As already mentioned, to present our approach more comprehensibly we divide its description into three parts. In the first part, i.e. in our *basic approach*, we consider attackers of type *Eve* only (Chapter 3). In the second part, i.e. our *fault-tolerant approach*, we add device failures and *Trudy* attackers (Chapter 4). As will be seen, the basic approach relies on a standard path search algorithm. These algorithms might use a lot of resources. This is the main reason for extending our approach in the third part, i.e. our *extended approach*. This extension does not rely on a path search algorithm, and therefore is even more suited for resource-constrained devices. Our extended approach is presented in Chapter 5.

3

Basic Key Distribution Scheme

As already mentioned, we have developed three approaches for a secure key distribution scheme for sensor and actuator networks. In this chapter, we present our basic approach. This approach guarantees the secrecy of a newly established key between two arbitrary devices in the presence of Eve-type attackers. The basic approach is the basis for our two extended approaches that are discussed in Chapter 4 and 5, respectively.

The remainder of this chapter is structured as follows. In the next section we give a short overview of the basic approach. After that we describe in Section 3.2 our algorithms for choosing the initial key sets, i.e. the key pre-distribution. In addition we prove the correctness of these algorithms. The corresponding key establishment algorithm is presented in Section 3.3. In Section 3.4 we analyze different methods for the out-of-band key pre-distribution. Different path search algorithms are discussed in Section 3.5. We conclude the chapter with a short summary.

3.1 Overview

Overall, our objective is to provide secure communication between any two devices of the network. We assume that our physical neighborhood graph G_N is connected, i.e. not partitioned and due to a routing layer the communication graph G_C is fully connected, thus any device can communicate with every other device in the network. In order to provide *secure* communication, we need a shared key between any device-pair of the network. Some keys will be available due to the pre-distribution of keys, others need to be established on demand. Establishing a shared key between any pair of devices must be secure, i.e. without giving an eavesdropping attacker (Eve) the possibility to learn the newly established key.

The fundamental idea of our key distribution scheme is the usage of non-random key distribution in order to induce some specific properties in the created key graph G_K. Due to our key graph construction algorithm we can guarantee that the created key graph is s-connected with s being the security level as defined in Section 2.4. This means, that in mathematical terms there exist at least s node-disjoint paths between any two nodes in the key graph. The key establishment then uses these node-disjoint paths to transport shares of a new key. The new key is comprised of all transported shares. Thus the new key remains secure as long as the attacker is not able to compromise all paths.

3.2 Key Graph Construction

In this section we describe our approach for the key pre-distribution. According to Section 2.2, the size of a sensor and actuator network may change dynamically. Therefore, a suitable key distribution scheme must include algorithms for adding and removing devices from the network at runtime. To do so, we provide an algorithm for the dynamic construction of a suitable key graph. Our key graph construction algorithm is comprised of two parts. In the first part we specify how a device is added to the network. In the second part we present how to remove a device from the network. Note that in this chapter the removal of a device is always assumed to be a controlled shutdown of a device. That means, that the device to be removed has the necessary time to perform the removal algorithm before leaving.

3.2.1 Adding a New Device to the Network

When a new device is added to the network we pre-distribute at least s keys between the new device and s devices which are already part of the network, with s being the security level. If the network does not yet have s devices, we need to provide a pre-distributed key to all of the devices which are already part of the network.

Using our formal representation of the key graph, we introduce new nodes into the graph incrementally. That is, whenever a new node is added to an already existing key graph at least s edges from this new node into the existing key graph need to be provided. The key graph construction algorithm is given in Algorithm 3.1.

According to Algorithm 3.1, for the first $s+1$ nodes, the network will be represented by a fully connected graph and for each additional node introduced a key needs to be securely exchanged with s other devices in the network, i.e. s new edges from the new node to previously existing nodes will be added. Note that whenever a key is exchanged (or pre-distributed) with this algorithm between two devices the key is stored in the main key store of that device, i.e. the

Algorithm 3.1 Basic key graph construction – adding a node

1: Given a graph $G_K = (V, E_K)$ with $n = |V|$ with nodes $v_i \in V$ and a new node v_{n+1}
2: $V := V \cup \{v_{n+1}\}$;
3: **if** $s \geq n$ **then**
4: **for** $i = 1$ to n **do**
5: $E_K = E_K \cup \{v_{n+1}, v_i\}$; // provide a new pairwise key securely
6: **end for**
7: **else**
8: $V' :=$ random subset of $V - \{v_{n+1}\}$ with $|V'| = s$;
9: **for** $i = 1$ to s **do**
10: $E_K = E_K \cup \{v_{n+1}, v_i\}$ with $v_i \in V'$; // provide a new pairwise key securely
11: **end for**
12: **end if**

KeyStore as introduced in Section 2.5. Hence each edge in the key graph will be represented by a key in the (main) *KeyStore* of two devices.

For an example, consider Figure 3.1: For a desired security level of $s = 3$, steps (a) through (d) build a fully connected graph. As soon as there are more than s nodes already in the graph, new nodes will be connected to the graph by exactly s new edges (steps (e) and (f)).

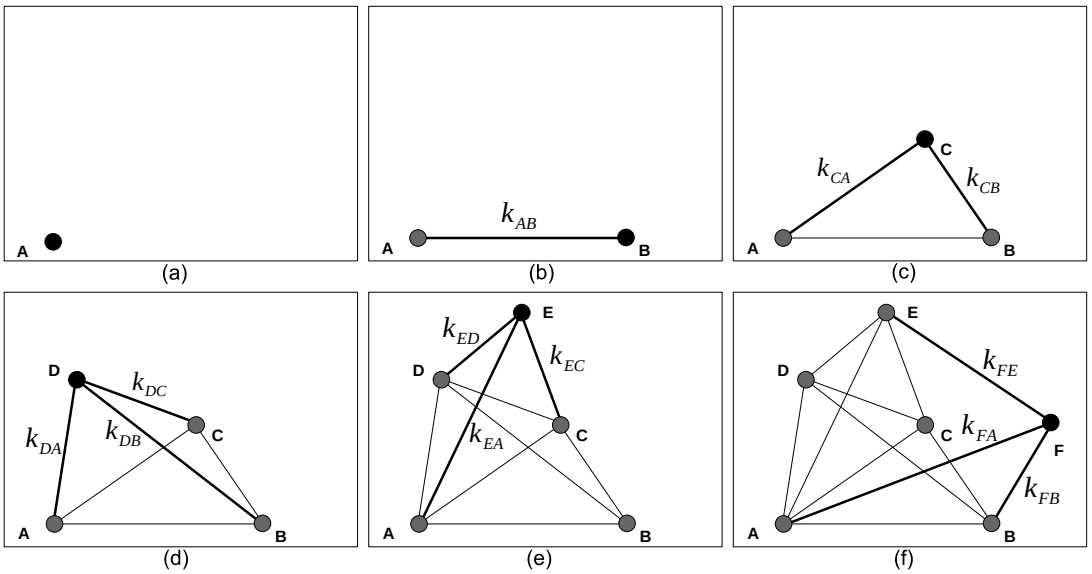


Figure 3.1: Introducing new nodes to the network graph ($s = 3$)

In detail, when introducing the first device (A) into the network no key needs to be provided, since there are no other devices in the network yet (Figure 3.1(a)). When introducing device B into the network we need to provide it only with a key to A since there is only one ($< s$) device

(A) in the network (Figure 3.1(b)). The same holds true for device C: we need to provide it with a key to A and B (Figure 3.1(c)). Also D needs to be provided with a key to all other devices which are already in the network, namely A, B and C (Figure 3.1(d)). From now on, there are at least four devices in the network, i.e. more than $s = 3$. Thus any new device needs to be provided only with $s = 3$ keys to some devices which are already in the network. We need to provide device E with keys to three random devices out of the four devices being already in the network, namely A through D. Here we chose devices A, C and D (Figure 3.1(e)). Also for device F only $s = 3$ keys to other devices need to be provided. In this case devices A, B and E were used (Figure 3.1(f)), but any other combination of three devices out of the existing network would do as well.

As we are going to show in Section 3.2.3 this construction algorithm yields an s-connected key graph. Due to this property secure key establishment between any devices which do not share a key yet can be guaranteed as long as the attacker has compromised fewer than s devices. Furthermore, the network size is not constrained in any way since we can add devices as needed.

3.2.2 Removing a Device from the Network

In the last paragraph we described our construction algorithm which produces an s-connected key graph. However, as can be easily seen when removing a device from such a network, the s-connected property of the underlying key graph might be destroyed. Thus we need an algorithm for removing devices from the network without destroying the s-connectivity.

We assume – for now – that the removal of a device occurs as a controlled shutdown, i.e. a device has the time to announce its impending departure from the network and make all necessary arrangements. We drop this assumption in Chapter 4 and extend our approach to cope with device failures.

Informally, our solution is based on pretending that the device that is to be removed (A) had never been there in the first place. As we saw in the last section, whenever a device B is added to the network it needs to establish keys with a set of s devices that are already present. If B is introduced after A, there is a certain chance that A was included in this set. Without A being in the network, we would have chosen a different set of devices without A. Therefore, to remove A, we have to replace it in the set.

Formally, let the key graph under consideration be $G_K = (V, E_K)$ with $V = \{v_1, v_2, \ldots v_n\}$, where a node v_i is the i-th node added to the graph according to our construction. We assume, that there are at least $(s+1)$ nodes in the graph, i.e. $|V| \geq (s+1)$. Otherwise, our graph construction algorithm ensures that the key graph is fully connected. Removing a node and all its corresponding edges from a fully connected graph results in a graph that is still fully connected. Therefore, removing a node cannot violate the connectivity.

Let the node that is to be removed from the graph be v_j. According to our graph construction algorithm, when adding v_j to the graph, s edges were established with already existing nodes. We call the set of these nodes the *parents-set* $\mathcal{P}_{v_j}$. $\mathcal{P}_{v_j}$ is defined as $\mathcal{P}_{v_j} := \{v_i | (i < j) \wedge (v_j, v_i) \in E_K)$ with $|\mathcal{P}_{v_j}| = s$. Informally, all nodes in $\mathcal{P}_{v_j}$ have been introduced *before* v_j and share a key with it. Furthermore, we define a set of nodes, containing nodes that were added to the graph after v_j and that, when they were introduced, established an edge with v_j. We call this set the *children-set* $\mathcal{C}_{v_j}$, and all nodes in $\mathcal{C}_{v_j}$ are younger than v_j. Figure 3.2 shows these sets.

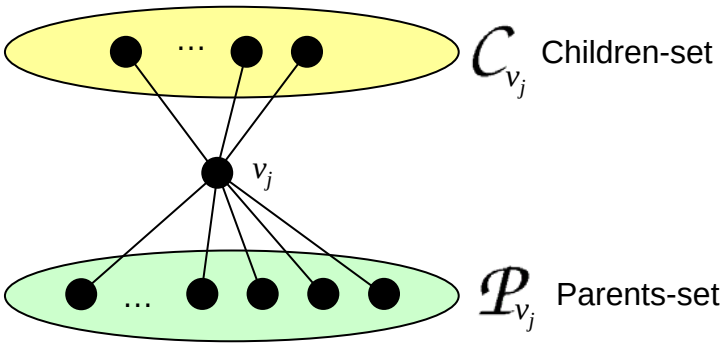


Figure 3.2: Parents ($\mathcal{P}_{v_j}$) and children ($\mathcal{C}_{v_j}$) sets

If node v_j is removed, each node $v_i \in \mathcal{C}_{v_j}$ will be missing one of its original edges, possibly violating the s-connected property. In order to replace this edge, we need to add a new edge with a node that did already exist when v_i was added to the graph. The algorithm is shown in Algorithm 3.2 using our formal definition of the key graph.

Algorithm 3.2 Removing a node from the key graph G_K with global knowledge

1: Given a graph $G_K = (V, E_K)$ and a node to be removed $v_j \in V$
2: **for all** $v_i \in \mathcal{C}_{v_j}$ **do**
3: Find a v_p where $(p < i) \wedge ((v_i, v_p) \notin E_K) \wedge j \neq p$
4: $E_K := E_K \cup (v_i, v_p)$
5: **end for**

The algorithm iterates over all nodes v_i in C_{v_j} in order to replace v_j in their parents-sets $\mathcal{P}_{v_i}$. To do so, we need to find a node v_p which was introduced before v_i (i.e. $p < i$) such that there is no edge (v_i, v_p) in E_K. After that, we add a new edge (v_i, v_p) to E_K.

This algorithm, however, uses global knowledge about which nodes are present in the graph, in order to choose a suitable v_p. To gain such knowledge is a non-trivial task. Hence, we modify the algorithm to rely only on local knowledge about the graph structure. Due to the definition of the set $\mathcal{P}_{v_j}$ we know that $k < j$ for each node $v_k \in \mathcal{P}_{v_j}$, i.e. every node in $\mathcal{P}_{v_j}$ was introduced before v_j. In addition, due to the definition of $\mathcal{C}_{v_j}$, we know that $j < i$ for each node $v_i \in \mathcal{C}_{v_j}$. Therefore, we can deduce transitively that $k < j < i$ and thus $k < i$. That means, that all nodes in

the parents-set of v_j were introduced before v_i and fulfill our first requirement in line 3. Hence, we choose v_p from $\mathcal{P}_{v_j}$. To fulfill the second requirement, we must select v_p such that $(v_i, v_p) \notin E_K$. Such a node can always be found in $\mathcal{P}_{v_j}$, because $(|\mathcal{P}_{v_i}| = |\mathcal{P}_{v_j}| = s) \wedge (v_j \in \mathcal{P}_{v_i}) \wedge (v_j \notin \mathcal{P}_{v_j})$. The modified algorithm is shown in Algorithm 3.3.

Algorithm 3.3 Removing a node from the key graph G_K with local knowledge

1: Given a graph $G_K = (V, E_K)$ and a node to be removed $v_j \in V$
2: **for all** $v_i \in \mathcal{C}_{v_j}$ **do**
3: Find a v_p where $(v_p \in \mathcal{P}_{v_j}) \wedge (v_p \notin \mathcal{P}_{v_i})$
4: $E_K := E_K \cup (v_i, v_p)$
5: **end for**

The only difference to Algorithm 3.3 is in line 3. Instead of generally searching for a node introduced before v_i we consider only nodes v_p from the set $\mathcal{P}_{v_j}$ such that there is no edge (v_i, v_p) in E_K.

Using this algorithm, we can remove all nodes v_j with $j > s$. However, if $j \leq s$, i.e. v_j is among the first s nodes in the graph, its parents-set as defined before contains less than s devices. Therefore, we cannot guarantee, that Algorithm 3.3 finds a node v_p (line 3). Thus, for such nodes, we redefine $\mathcal{P}_{v_j} := \{v_i | 1 \leq i \leq (s+1) \wedge i \neq j\}$. Using this modified parents-set we can now replace the missing edges for all nodes $v_i \in \mathcal{C}_{v_j}$ with new edges to a node from $\mathcal{P}_{v_j}$ as shown before. After this is done, v_{s+2} will share links with all s nodes in $\mathcal{P}_{v_j}$, therefore $\mathcal{P}_{v_j} \cup \{v_{s+2}\}$ will be fully connected. Also, all $v_j \in \mathcal{C}_{v_j} - \mathcal{P}_{v_j}$ will have a new edge with a node in $\mathcal{P}_{v_j}$, so the *s-connected* property is sustained.

In order to realize this algorithm in a sensor or actuator network, each device only needs to know its own parents-set and that of the leaving device. Using the data structures from Section 2.5 this is done by implementing the *KeyStore* as a linear list. Thus the parents-set is simply the first s keys from the *KeyStore* and can be retrieved using the already introduced methods of *KeyStore*. To identify the parents-set and the children-set is also the reason for introducing two key stores, i.e. the main and the establishment key store. By splitting these key stores it is always possible to determine which keys represent edges in the key graph and which not. Clearly this could also be done by introducing an additional flag, but we choose for didactical reasons to work with two key stores. With this in mind we develop two Procedures 3.1 and 3.2, which we describe in the following.

Whenever a device needs to perform a controlled shutdown, i.e. leave the network, the **shutdown** procedure is executed (Procedure 3.1). Each device which shares a key with the leaving device might need to replace a key in order to sustain the s-connected property of the key graph after the leaving device has shutdown. Therefore, the leaving device first notifies all these devices about its intentions by securely sending a ”LeavingIntention”-message to them. This

```
1: procedure shutdown()
2: ParentsSet := KeyStore.GetParentSet(); // first s devices from the KeyStore, set 𝒫)
3: for all Device ∈ KeyStore.GetAllIDs() do
4:     SecureSend(Device, LeavingIntention, {ParentsSet});
5: end for
6: wait for all ACKs;
7: power off;
```

Procedure 3.1: Leaving the network: shutdown procedure

message contains the parents-set of the leaving device to enable the other devices to find a suitable replacement.

After the ”LeavingIntention”-message has been sent, the shutdown procedure waits for an acknowledgment from all contacted devices. This is necessary because the other devices may need the presence of the leaving device to replace the key. If the device leaves prematurely, the key graph could loose its s-connectivity and no further keys could be established securely.

```
1: onReceiveLeavingIntention(Source, {SourceParents});
2: if Source ∈ KeyStore.GetParentsSet() then
3:     Candidates := SourceParents − KeyStore;
4:     Device := pick randomly one from Candidates;
5:     establish a new shared key with Device and store in EstablishmentKeyStore;
6:     SecureSend(Source, ACK, {});
7:     KeyStore.ReplaceKey(Source, Device, EstablishmentKeyStore.GetKey(Device));
8:     EstablishmentKeyStore.DeleteKey(Device);
9:     SecureSend(Device, NewChild, { });
10: else
11:     SecureSend(Source, ACK, {});
12:     KeyStore.DeleteKey(Source);
13: end if
```

Procedure 3.2: Leaving the network: repairing the key graph

Any device receiving a ”LeavingIntention”-message must perform the procedure described in Procedure 3.2. First, the device checks if the message comes from a device in its own parents-set (line 2). If this is not the case, it can simply send an acknowledgement to the leaving device and delete the respective key. Otherwise it needs to replace the key in order to sustain the s-connected property of the underlying key graph. As already mentioned, the ”LeavingIntention”-message contains the parents-set of the leaving device. The receiving device needs to determine a device with which it does not share a key yet and which is in the parents-set of the leaving device (line 3). When such a device is found, a new key is established to it using the key establishment algorithm described in the next section. Whenever a

new key is established with the key establishment algorithm, this key is stored in the secondary key store, i.e. the *EstablishmentKeyStore*. After establishing the new key, the receiving device sends its acknowledgement to the leaving device. After this, the key to the leaving device in the main key store is replaced with the newly established key from the secondary key store. Furthermore, the key is removed from the *EstablishmentKeyStore* since this key becomes part of the main key store, i.e. it is an edge in the key graph. In order to tell the other device, that this key should be moved to the main key store the ”NewChild”-message is send next. Note that it is important to send the ”ACK”-message before replacing the key – otherwise sending securely would fail (see line 6 and 12).

```
1: procedure onReceiveNewChild(Source,{ });
2: // update key status
3: KeyStore.AddKey(Source, EstablishmentKeyStore.GetKey(Source));
4: EstablishmentKeyStore.DeleteKey(Source);
```

Procedure 3.3: Moving a key to the main key store

The reception of a ”NewChild”-message informs a device that the sender of this message was involved in a leaving algorithm and the key to this sender needs to be moved from the secondary key store to the main one. The details are given in Procedure 3.3. After executing this procedure the key with the sending device is moved to the main key store and therefore, formally we added a new edge to the key graph.

After all devices have performed Procedure 3.2 and the leaving device received acknowledgements from all contacted devices, the key graph has been modified successfully. The leaving device has been removed from it and can therefore safely be shutdown.

For an example, consider Figure 3.3. Device C is to be removed. It sends a message to all neighbors announcing its impending departure and includes the first s devices it has pre-distributed keys with (A,B,D). Devices A,B and D already share keys with each other, so they will not need to act. Device E, however, learns that it needs to establish a new key and the only option is to do so with B. A new key is established between E and B. After all other devices have acknowledged, C can then leave the network.

3.2.3 Proof of Correctness

In the following we show that the construction of the key graph described in Section 3.2.1 will always result in s disjoint paths between any pair of nodes for all key graphs (V,E_K) with $|V| > s+1$.

We use the following definition from graph theory:

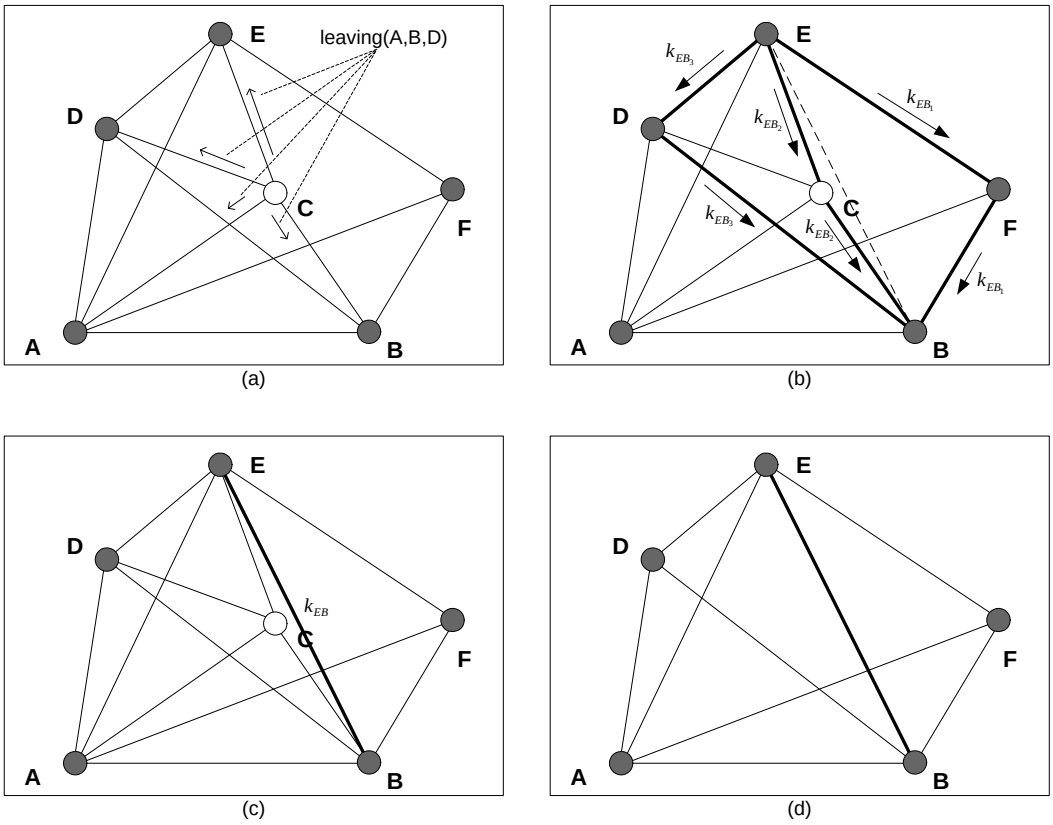


Figure 3.3: Removing nodes from the network graph ($s = 3$)

Definition 3.1 (k-connected graph) *A graph $G = (V,E)$ is said to be* k-connected *if and only if for any set $W \subseteq V$ with $|W| < k$, the subgraph induced by $V - W$ is still connected.*

Theorem 3.1 (Menger's Theorem [Men27]) *In a k-connected graph, there always exist k node-disjoint paths between any pair of non-neighboring nodes.*

PROOF See, for instance, [Har95]. ■

Theorem 3.2 *Any graph $G_K = (V,E_K)$ as constructed using Algorithm 3.1, with $|V| \geq s$ will be s-connected.*

PROOF Induction over $|V|$:

Induction Statement Adding a new node to a given s-connected graph $G = (V,E_K)$ with $n = |V| > s$ by connecting the new node with at least s edges to the existing graph results in a new graph with $|V| = (n+1)$ which is also s-connected.

Induction Start For $|V| = (s+1)$, the graph will be fully connected, so it follows trivially that the graph is also s-connected.

Induction Step Now consider an already s-connected graph $G = (V, E_K)$ with $|V| > (s+1)$. We add a new node v using Algorithm 3.1. We have s new edges into the existing graph. Since the original graph was already s-connected, after adding new edges, it will still be s-connected. And, in order to disconnect the new node, we would need to remove at least s other nodes. So s-connectivity cannot be violated by detaching the new node v either. Hence, due to Definition 3.1 of a k-connected graph also the graph $G = (V, E_K)$ with the additional node v is s-connected. ■

Theorem 3.3 *In any graph $G_K = (V, E_K)$ as constructed using algorithm Algorithm 3.1, with $|V| \geq s+1$, there will always be s node-disjoint paths between any pair of nodes.*

PROOF Follows directly from Theorem 3.1 and 3.2. ■

It remains to be shown that the algorithms for removing a device from the network do not destroy the s-connected property of the key graph.

Theorem 3.4 *Any s-connected graph $G_K = (V, E_K)$ as constructed using Algorithm 3.1 will still be s-connected after the execution of Algorithm 3.2 or Algorithm 3.3.*

PROOF The correctness of the removing algorithm follows directly from the correctness of the key graph construction algorithm. Executing one of the removing algorithms on a key graph G_K constructed with Algorithm 3.1 results in another key graph G'_K which could have been constructed using the key graph construction algorithm (by design of the removing algorithms). Hence, also the resulting key graph G'_K is s-connected. ■

3.3 Key Establishment

As already stated, our key distribution scheme comprises of the key pre-distribution algorithm and the key establishment algorithm. In this section we describe the key establishment algorithm.

Obviously, there is no need to establish additional shared keys for networks with up to $(s+1)$ devices, since the corresponding key graph will always be fully connected. Thus, for the following discussion we only consider networks with more than $(s+1)$ devices, i.e. $|V| > (s+1)$.

We first describe how two devices can establish a new shared key between them. In order to establish an l-bit key, a device randomly generates s l-bit shares $k_1, \ldots k_s$, and sends them over

s device-disjoint paths (i.e. paths that do not share common devices) to the destination device (Figure 3.4). On each hop of a path, the key share is transmitted in an encrypted fashion using the existing appropriate shared key. The final key k is then calculated by $k = k_1 \oplus k_2 \oplus \ldots \oplus k_s$, where $\oplus$ is the bitwise XOR operation. This approach is also used in [CPS03, ZH99, Gon93].

It should be clear that without having access to all key shares, an attacker cannot recover the key. To compromise the newly established key an attacker needs to subvert at least s devices (one on each path) because the key shares are communicated over s device-disjoint secure paths. Note that due to the pairwise symmetric keys devices are authenticated against each other and therefore a single attacker device cannot be a member of multiple paths.

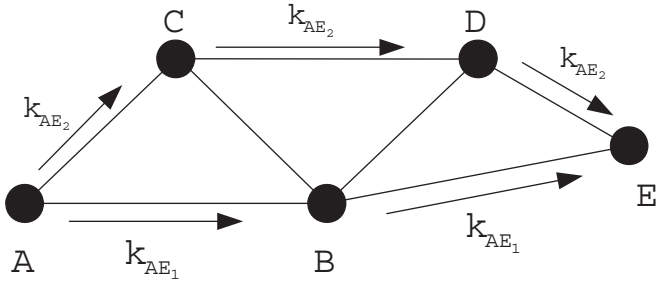


Figure 3.4: Key establishment $(s = 2)$

Our algorithm for key establishment is given in Procedures 3.4 and 3.5. A device which needs to establish a shared key with another device first needs to discover at least s node-disjoint paths in the key graph. These different paths can be discovered using standard methods, e.g. through bounded depth-first search. Since such a path discovery is not trivial on very resource-constrained devices, a discussion about different methods will be given in Section 3.5. For now we assume that the operation `DiscoverNodeDisjointPaths` returns s device-disjoint paths (line 2 in Procedure 3.4). After that, for each one of the s paths a random key-share is generated and securely sent to the next node on the path. Finally, the new key is stored in the secondary key store, i.e. *EstablishmentKeyStore*. As already mentioned, we store newly established keys in the secondary key store in order to distinguish later between the key pre-distribution set and the keys which are established through the key exchange protocol.

The transport and routing of the key shares is based on source routing: A device which receives a "KeyShareTransport"-message, first removes the first entry of the *Path* being sent along (line 2 in Procedure 3.5). This element is its own device ID, since it received this message. The resulting *RestPath* contains the remaining devices along the path. After that, the device replaces the *Path* in the "KeyShareTransport"-message with the *RestPath* and forwards the updated message securely to the next device on the path. In case the *RestPath* is empty, the packet reached

```
1: procedure KeyEstablishment(Partner);
2: Paths := DiscoverNodeDisjointPaths(Partner);
3: // Paths is a two dimensional array containing all found paths;
4: // First dimension is the i-th path, second dimension is the device ID on the path
5: // e.g. Paths[i][3] is the device ID of the third device on the i-th path
6: for i = 0 to s − 1 do
7:    KeyShare := random(KEY_LENGTH);
8:    EstablishmentKeyStore.UpdateKey(Partner, KeyShare)
9:    NextHop := Paths[i][0]
10:   SecureSend(NextHop, KeyShareTransport, {MyDeviceID, KeyShare, Paths[i]});
11: end for
```

Procedure 3.4: Initiating key establishment

its destination and the key share needs to be processed, i.e. xor'ed with all the other received key-shares.

```
1: procedure onKeyEstablishmentTransport(Source, InitiatorID, KeyShare, Path[]);
2: RestPath := Path - { Path[0] }; // Delete the first element
3: if |RestPath| > 0 then
4:    // we are just transporting the share, thus we need to forward
5:    SecureSend(RestPath[0], KeyShareTransport, {InitiatorID, KeyShare, RestPath});
6: else
7:    // we are the receiver of this share, thus store the share
8:    EstablishmentKeyStore.Update(InitiatorID, KeyShare);
9: end if
```

Procedure 3.5: Receiving key establishment

3.4 Bootstrapping: Creating the Initial Key-Graph

In the last sections we described which key sets need to be stored on the devices and how two devices can establish a new key if they don't share one yet. As already mentioned the initial keys need to be distributed to the devices over a secure channel, i.e. out-of-band . In the following we describe some possibilities to do so.

3.4.1 Static Pre-Distribution

The most common method used when deploying a sensor net in the field is the key pre-distribution through the manufacturer. In this case the manufacturer prepares all devices of the

network before deploying it in the field. In our case that means that according to Algorithm 3.1 each device gets loaded with the appropriate set of keys before the devices are brought in place. Thus, after deployment the initial key graph is already established and any two arbitrary devices may establish a new key on demand based on the initial key sets.

The advantage of this approach is its simplicity. The burden lies completely with the manufacturer who needs to store the appropriate initial keys on each device. However, the major drawback is, that with this method it is impossible to add new devices after the network is deployed in the field. Thus this approach does not cope with our requirement of dynamic network sizes (Section 2.2).

Therefore, this method can be used when deploying a sensor network in the field for instance for wildlife surveillance where the size is known from the beginning, and it does not change over time, i.e. a static sensor network. It is unsuitable for bootstrapping a sensor or actuator network in a users home for instance for home automation or pervasive computing environment. Here it is often needed that devices are added to an already working network, i.e. a dynamic network.

3.4.2 Configuration Device

Another method for storing the initial key set on the devices is the usage of a configuration device. With this method a unique configuration key needs to be initially loaded on each device. This can be done for instance by the manufacturer of the device. Therefore, when a user buys a new device, the configuration key must be provided along with it, e.g. in the instruction manual. Clearly, also a way of changing this configuration key could be provided in order to allow the user to change the configuration key. Alternatively the manufacturer does not provide a configuration key and the user needs to set one up before the device becomes operational.

The software on each device allows to store and to delete keys from the devices if the communication is secured (i.e. encrypted and authenticated) with the configuration key. In order to configure a new device, i.e. store new keys on it and therefore add it to the network we store its configuration key on the so called configuration device. With the aid of the configuration key, the configuration device can now communicate securely with each individual device and provide them with the appropriate keys.

The configuration device does not suffer from the same resource limitations as the devices themselves, and can therefore store one configuration key to each device in the network. Clearly, this device must be kept in a secure place, since if it falls into the hands of an attacker the network is completely compromised. However, this is not a issue, since the configuration device is only needed for configuration purposes, i.e. adding a device to the network. Therefore most of the time this device can stay in a secure environment like a safe for instance.

As an example consider the usage of a secure smartcard. The user stores all configuration keys in an encrypted file somewhere on his personal computer. He additionally posses a secure smartcard with a 2048 bit RSA key pair. The file is encrypted with the public RSA key and can therefore only be decrypted with the aid of the corresponding private key - which is safely stored on the smartcard. Thus, only the smartcard needs to be kept in a secure place. Furthermore it would be wise to keep also some backups of the encrypted files in different places in order not to loose it, since it is crucial for configuring the network. In this example the actual configuration device would be the personal computer which needs to be equipped with a smartcard-reader.

The main advantage of this method is that it copes with dynamic network sizes: Whenever the user buys a new device, he registers it with the master device and is then able to configure it at will. The only drawback of this method is that if the user looses the configuration keys he looses control over the network.

Assuming it has been taken care of storing the configuration keys safely this method is suitable for large sensor networks in the field as well as the pervasive computing environment in the users home.

3.4.3 Physical Contact

Following [SA99] physical contact is one of the most secure ways to establish a new secret between two devices. We can use this finding in order to bootstrap new devices as follows: Initially a password (e.g. some numbers) needs to be setup for a new device before it becomes operational. This password is only known to the user and needs to be entered directly at the device. Thus each device needs the appropriate input capabilities.

For loading a new device with a new key in order to become part of the network, this device needs to be brought in physical vicinity of a device which is already part of the network. The two devices need to be connected physical e.g. through a wire. After that, the user needs to enter the password on each device and then initiate the key generation. The password is only needed to authenticate the user against the devices in order to put those devices in key pre-distribution mode, i.e. they initiate a protocol over the physical connection in order to exchange a new unique key. When this is done, the two devices share a new unique key and can therefore be physically disconnected, since they are now able to communicate securely over the new key. In order to introduce a new device into the network according to our key graph construction algorithm (see Algorithm 3.1) this step of physically connecting the device to an already existing device from the network needs to be repeated s times.

The main advantage of this approach is that there is no additional hardware involved in the process of adding a device to the network, especially there is no master device. The user simply buys a new device, sets up his password (which might be the same for all devices) and

connects it physical to s other devices which he has already in his home. After that the device is integrated in his network and is able to communicate securely with any other device in this network. Thus, the simplicity of this method is another advantage. The drawback is that for a large s, i.e. for a high security level it may become cumbersome to connect a new device to s already existing devices. It may also be cumbersome if someone needs to add many devices at the same time, i.e. if someone builds almost the whole network from the beginning.

Thus, we can summarize that his method is suitable for a user who sets up his pervasive computing environment incrementally adding now an then a new device. It is unsuitable if a complete network e.g. a sensor network needs to be set up at the beginning for instance for bridge or wildlife surveillance.

3.4.4 Hybrid

In order to overcome the drawbacks of the above methods and to use only the advantages from each method a hybrid method for bootstrapping new devices can be used. For instance for the classical sensor network used for wildlife surveillance the manufacturer of the devices might store already an initial key set on the devices, thus when the network is deployed in the field it is operational from the beginning. Additionally the manufacturer stores a configuration key on each device. These configuration keys are provided to the user as well. Thus, with the aid of these configuration keys new devices may be added later at any time. Hence, we combined the advantages of the pre-distribution method with the advantage of the configuration device method.

In a pervasive computing environment the physical contact method could be combined with the configuration device method. This way the user may choose to introduce a new device into the network with the configuration device or by simply physically connecting it to s already existing devices.

Clearly, also other combinations are possible. Which method suits best depends on the situation and the actual requirements regarding security level, simplicity etc..

3.5 Path Search Protocols

In this section we analyze different possibilities for finding the s device-disjoint secure paths, which are needed by the key establishment protocol. Generally, we can distinguish between two classes of algorithms: first algorithms which use global knowledge about the underlying key graph and second algorithms which use local knowledge only, i.e. distributed algorithms. In the following we describe these two classes.

3.5.1 Global Knowledge Algorithms

Most algorithms from graph theory for finding node-disjoint paths in a given graph are in this class. These algorithms use a graph $G = (V,E)$ as input and return the disjoint paths. Since these algorithms operate on the entire graph of the network, the minimum memory requirement is to store the entire graph on each device. Hence, the lower bound for memory consumption is $|V| + |E|$.

The vertex-disjoint Menger problem i.e. finding s node-disjoint paths in a s-connected graph is equivalent with finding the maximum flow or the minimum cut, also referred to as the max-flow-min-cut problem [Har95]. The best known results for solving this problem where achieved by Goldberg and Rao [GR98]: They achieve a runtime of $O(min(|V|^{2/3}, \sqrt{|E|})|E|)$, which suggests that $O(|V||E|)$ is not the lower bound as it was considered before. An overview of developments for solving the maximum flow problem is given in [Gol98].

Algorithms for solving this problem are either augmenting path algorithms or push-relabel algorithms, which are shortly discussed in the following.

Augmenting Paths Algorithms

The basic principle of augmenting path algorithms was described by [FF57] with a runtime of $O(|V||E|)$. This algorithms works by finding and augmenting paths. However, the augmentation is not considered in the above runtime, and must therefore be added to the runtime of this algorithm. Based on the Ford and Fulkerson approach Edmonds and Karp presented an algorithm which includes the actual path augmentation algorithm by using the shortest path method [EK72]. This results in $O(|V||E|^2)$ time complexity. Further optimizations were achieved by Dinic [Din70] with a runtime of $O(|V|^2|E|)$ or in a later optimized version with McKay heuristics a runtime of $O(|E|^{3/2})$.

Push-Relabel Algorithms

The basic principle of a push-relabel algorithm for the max-flow-min-cut problem was presented by Goldberg and Tarjan [GT88] which runs in $O(|V|^2|E|)$. The optimized version runs in $O(|V||E| + |V|^2 log(|V|^2/|E|)$. Further optimizations are presented by Karzanov [Kar74] with runtime of $O(|V|^3)$ and by Cherkasky [Che77] with a runtime of $O(|V|^2\sqrt{|E|})$.

For some special cases like e.g. planar graphs there exist linear time algorithms [RLWW97]. However, with a security higher than two our graphs cannot be planar, hence these algorithms are not applicable.

Conclusion

We can conclude that a global view algorithm for locally calculating the s node-disjoint paths from a given graph $G = (V,E)$ runs with a minimum of $O(min(|V|^{2/3}, \sqrt{|E|})|E|)$ runtime with at least $O(|V| + |E|)$-space requirement.

A global knowledge algorithm additionally needs to gather global knowledge of the key graph in order to be usable for our approach. Gathering the key graph could be done e.g. by the usage of a link-state approach, which needs a minimum of $O(|E|)$ messages. Furthermore, the key graph needs to be updated whenever a new device is added or an old device is removed.

3.5.2 Distributed Algorithms

Distributed algorithms only use local knowledge on each device for their calculations, i.e. a device knows only about its direct neighbors. In order to find the s node-disjoint paths the devices need to communicate with each other and perform a distributed algorithm.

A basic approach for a distributed path search algorithm is based on a breadth first search with limited search depth. To detect all paths to a given target device a source device sends a search query to all its neighbors which in turn forward the query to all their neighbors until the search depth is reached. If the target device is reached, it sends back all paths used to contact it. The source device stores all received paths and selects node-disjoint paths from them. Note that in order to detect all paths, devices receiving a query that they have already seen earlier cannot simply discard the query but may have to forward it again. This leads to a much higher communication overhead compared with searching only one path. To limit the communication overhead, the path search depth can be bound to a certain factor, leading to the discovery of all paths up to the limited length. If the source device is able to detect enough node-disjoint paths in this subset of all paths, the search can be terminated. Otherwise, another search with a higher limitation factor must be initiated.

This algorithm has a communication complexity of $O(\Delta_{v_i}^d)$ with Δ_{v_i} being the degree of a node v_i i.e. the number of neighbors and d the search depth. In the worst case, d can grow up to $(|V| - 1)$ for some graph $G = (V,E)$, which leads to an exponential communication overhead. Concerning the space complexity of the approach, the source device needs to store all returned paths in order to compare them with each other for detecting the s node-disjoint paths. The length of the paths is only limited by the network size $|V|$, i.e. is in $O(|V|)$. Thus, the space complexity is $O(|V|)$. Due to its high space and communication complexity this approach is only suitable for small networks.

In order to increase the resilience of the network against link failures, a large number of algorithms to search for *edge-disjoint* paths has been proposed in the literature, e.g. [SAN90,

Bha97, KS98, MD01, LG01, GKvM03, SM03, IR84, XCX+04]. However, in the context of this work we need node-disjoint paths, which are not discovered by these algorithms.

Ogier and Shacham [OS89, ORS93] describe a distributed algorithm to find shortest pairs of node-disjoint paths based on the method of link-disjoint augmentation presented by [Suu74]. In [CKGLA90] this work was extended to find k node-disjoint paths of minimum total length. Sidhu et al. [SNA91] further improved this approach. The space complexity of these algorithms, i.e. the memory consumption on each device is $O((\Delta_{v_i} + D)\log(|V|W))$ bits, where Δ_{v_i} is the number of neighbors of node v_i, D is the depth of a shortest path spanning tree directed toward z, and $\log(W)$ is the number of bits required to represent a single link length $c(i \ldots j)$. The communication complexity is given with $O(|E||V| + |V|^2 D)$ [ORS93].

3.5.3 Discussion

A problem shared by the above approaches is that actually finding node-disjoint paths is not trivial, especially on memory constrained devices. In both classes of presented algorithms large networks will lead to long paths between nodes and require larger packet sizes to store the path. Algorithms using global knowledge use less communication overhead, but need more memory to store the key graph. Distributed algorithms have a higher communication overhead while keeping the memory consumption low. Hence, if there is enough memory available then the global knowledge algorithm might be a good choice. In our scenario with very resource-constrained devices the distributed algorithms are more suitable. Furthermore, our extended approach, which we present in Chapter 5, does not rely on a path search in order to establish a new key, and is therefore even more suitable for our scenario.

3.6 Summary

In this chapter, we have presented our basic approach for key distribution in sensor and actuator networks based on symmetric cryptography. Our algorithm guarantees the ability for an arbitrary pair of devices to exchange a key in a secure fashion, provided that the number of devices an passive attacker (”Eve”) is able to subvert is not higher than the security level s of the network. We achieved this without prior knowledge of the maximum network size. Additionally, we do not require any pre-configuration of the devices. The network can grow incrementally and also shrink if devices are removed in a controlled fashion.

However, the basic approach does not cope with device failures or active attackers., i.e. it provides only security against passive attackers. The extension to a fault tolerant approach is presented in the next chapter. Furthermore, as we saw in Section 3.5, the basic approach uses a path search algorithm in order to establish a new key. Since such algorithms might use many

resources we present in Chapter 5 another extension which does not rely on such a path search and is therefore more suitable for highly resource-constrained devices.

Our basic algorithm has been published in [WHC04] and presented at the IEEE conference CCNC 2005.

4

Fault Tolerant Key Distribution Scheme

In the last chapter we presented a key-distribution scheme that guarantees the secrecy of a key exchange as long as there are less than s subverted devices, where s can be chosen according to the actual security requirements. Device failures[1] and active attackers[2], i.e. *Trudy*, are not handled by the basic approach. Therefore, in this chapter we extend our basic approach towards a fault tolerant key establishment protocol.

We start in Section 4.1 with an overview of the damage inferred through device failures and active attackers. We then present in Section 4.2 an extended approach, which copes with device failures. After that we provide in Section 4.3 a detailed analysis of active attackers and provide possible countermeasures. Finally, we conclude this chapter with a short summary about the fault tolerant approach.

4.1 Analysis of the Basic Approach

Our basic key establishment algorithm from Chapter 3 guarantees the secrecy of a newly established key as long as *Eve* controls less than s devices. However, the s-connected property and therefore new key establishments can only be guaranteed if there are no device failures and no active attackers. Furthermore, also the secrecy of new keys cannot be guaranteed anymore in the presence of active attackers. In the following, we describe the problems which might occur.

[1] We consider only fail-stop device failures as defined in our system model in Section 2.1.

[2] In the context of this work we do not consider *Mallory*-type attackers.

4.1.1 Device Failure Impacts

A device failure may lead to problems with any algorithms presented so far, i.e. within the key graph construction algorithm and the key establishment algorithm. If a device fails during execution of an algorithm, it may fail to respond to some messages, thus leading to a device waiting infinitely. As an example consider Procedure 3.1 on page 53. In this procedure the leaving device waits for all acknowledgments of other devices. If one of these devices fails just before sending the acknowledgement, the device would wait forever. In order to avoid such situations we need to detect these cases and to react appropriately to them. Detection can be solved with timeout mechanisms. The appropriate reaction however depends on the situation. In our algorithms so far the only possible reaction is to terminate the communication with such a device. This means that the algorithms do not terminate.

Perpendicular to impacts on our presented algorithms is the impact of a device failure on the connectivity of the neighborhood graph G_N. As mentioned in Section 2.4, this graph is a graph whose nodes represent the devices and whose edges describe direct communication links between two devices. Imagine a sensor or actuator network built with our algorithms. The network neighborhood graph G_N is connected but very sparse. The failure of a single device could lead to the disconnection of the neighborhood graph, thus partitioning the network. If G_N is partitioned, also the key graph G_K is partitioned, thus rendering any presented algorithm unusable. A simple solution for this is to use the key graph construction algorithm as described in Algorithm 3.1 for the communication graph. When adding a device to the network, the device must be in communication range with a certain number r of devices. This produces a r-connected communication graph, which tolerates up to $r-1$ device failures without partitioning the network.

For the rest of this work, we assume that we have a sufficiently connected neighborhood graph. Thus, as stated in Section 2.1, communication between two arbitrary devices is always guaranteed.

In the following we analyze the impacts of device failures with respect to each of our algorithms presented so far in detail.

Impacts on the Key Establishment Algorithm

According to Procedure 3.4 on page 58 if a device needs to establish a new key with another device, it first discovers s device-disjoint paths to the target and then sends the s key-shares to the target device over the discovered paths. If a failed device is part of one secure path the corresponding key-share will not reach the target device. Due to timeout mechanisms the initiating and the receiving device may be aware of the situation, but might not be able to

react. This is simply because there is no other secure path available. Hence in the presence of a device failure the key establishment algorithm might not be executed correctly and the key establishment cannot be guaranteed anymore.

Impacts on the Key Graph Construction Algorithms

When adding a new device to the network according to Algorithm 3.1 we need to exchange s keys out-of-band (see also Section 3.4) with devices which are already in the network. If one of the chosen devices fails, adding the new device to the network will not succeed. However, this situation can be detected, e.g. through timeouts. After detecting the situation another set of s devices without the failed device can be chosen. Hence this situation can be easily solved.

When removing a device from the network, i.e. performing the controlled shutdown, a failed device which is in the children-set of the leaving device will not react according to Procedure 3.2. Here we need a timeout mechanism, since the leaving device would wait infinitely for the failed device to respond. With such a timeout mechanism the leaving device will simply give up waiting after some time and leave the network. Even if the failure does not directly hinder the controlled shutdown procedure, the connectivity of the resulting key graph cannot be guaranteed anymore - which in turn leads to problems with the key establishment.

The same holds true if a device fails without performing the controlled shutdown procedure. All devices from the children-set of the failed device will not be informed about the failure of the device, thus not performing the graph modifications necessary in order to keep the s-connectivity of the underlying key graph. Even if the devices from the children-set of the failed device would have some possibility to detect the failure they cannot act appropriately: First they need the information about the parents-set of the failed device, which would have been sent by the failing device. Second and even worse, even with this information it could not be guaranteed that they can establish a new key with a replacement device, since the failed device might be on a needed secure path. Therefore, also in this case the s-connectivity of the underlying key graph cannot be guaranteed anymore.

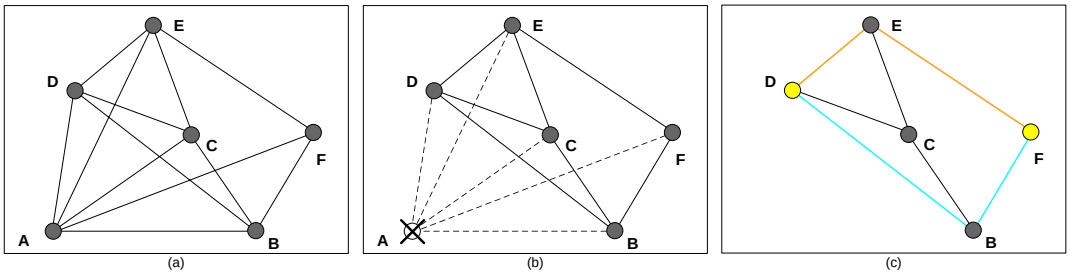


Figure 4.1: Failed device destroys graph connectivity ($s = 3$)

As an example consider Figure 4.1. In this network a security level of $s = 3$ was used, hence the connectivity of the key graph is three (see Figure 4.1(a)). After some device A fails and does not perform the controlled shutdown procedure (Figure 4.1(b)). The resulting graph has no longer a connectivity of three. This can be easily seen in Figure 4.1(c) since there are only two paths between device D and F available.

4.1.2 Active Attacks (*Trudy* and *Mallory*)

Even more damage can be done if a device is compromised by an active attacker. A device subverted by *Trudy* interacts with the algorithms with the deliberate intention to gain access to secrets. Furthermore, a device subverted by *Mallory* pursues the intention of hindering any new key establishments. To do so it can either destroy the s-connected property of the key graph or preventing the establishment of new keys directly by falsely interacting with the key exchange algorithm. In the following we sketch the basic possibilities of these two attackers. We provide a detailed analysis and possible countermeasures in Section 4.3.

Trudy's Attack Possibilities

One possibility for *Trudy* is to use the key establishment algorithm in order to mount a man-in-the-middle attack as depicted in Figure 4.2. She therefore prepares key-shares which seem to come from device A and sends them (over device-disjoint paths) to device B. After that she also sends to device B the key-shares which seem to come from device A. Devices A and B will believe that they are engaged in a correct key establishment algorithm: For each device it seems like the other device initiated the key establishment algorithm. After the successful key establishments (*Trudy*$\rightarrow A$ and *Trudy* $\rightarrow B$) also *Trudy* will know the new key and can therefore listen to all future messages exchanged between device A and B. This attack is possible since the key-shares are not authenticated – *Trudy* is able to forge the sender of the key shares.

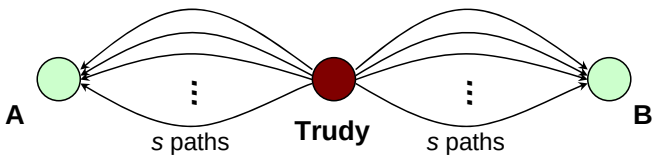


Figure 4.2: Man-in-the-middle attack

The second possibility of *Trudy* also involves misusing the key establishment algorithm in order to mount a Sybil-attack as depicted in Figure 4.3. In this attack *Trudy* is able to introduce herself multiple times into the network by establishing keys from non-existent virtual IDs (Trudy-1, Trudy-2 ...) which are controlled by her. After that she only has to intercept the key-shares

which are routed over her multiple virtual devices in order to gain access to new secrets. As an example consider Figure 4.3. *Trudy* subverted device G. After that, *Trudy* created a virtual identity named *Trudy-1* and initiated the key exchange between every device in the network and this virtual ID. Hence, after this the virtual ID of *Trudy* shares a key with every device in the network. *Trudy* could do the same with device G – for the sake of simplicity we show in the figure only the key to device A. Now imagine, that device A must establish a key with device F. Two disjoint paths between are $(A,$*Trudy-1*$,F)$ and (A,G,F). Hence, *Trudy* is part of both paths even though she subverted only a single device.

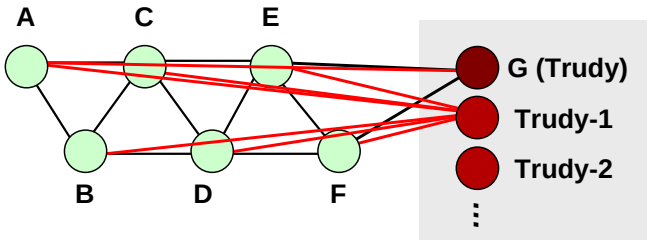


Figure 4.3: Sybil attack

More details and countermeasures about the two above attacks are given in Section 4.3.1.

Mallory's Attack Possibilities

Even though we do not consider *Mallory*-type attacker in the context of this work, we give a brief overview of *Mallory*'s possibilities. *Mallory* could interfere with the normal operation of the algorithms in even more ways. Imagine the following situations:

The subverted device might forward altered key-shares (see Section 3.3). As a result, the two participating devices will not share a common key, thus rendering the key establishment algorithm useless. Clearly, in order to do so the subverted device needs to be on the secure path between the two devices.

A subverted device might initiate the removal algorithm as described in Section 3.2.2, but send a false parents-set to its neighbors. As a result, the neighbors would establish new keys to the wrong devices, possibly destroying the s-connected property.

Furthermore a device under the control of *Mallory* might decide not to respond in a correct manner when the device gets a "LeavingIntention"-message. For instance it could simply send an "ACK" without performing the necessary operations.

Generally speaking, *Mallory* will use any chance given in order to hinder the successful key establishment between two uncompressed devices. She will try to decrease the connectivity of the underlying key graph by inserting fake messages or denying service.

More details about the above attacks are given in Section 4.3.2.

4.1.3 Conclusion

The main foundation of the basic key establishment algorithm is the s-connectivity of the underlying key graph. As long as the graph is s-connected, devices can be added and removed, new keys can be established, and Eve attackers can be tolerated. However, in the presence of device failures or active attackers, the basic approach cannot guarantee the s-connectivity of the key graph.

As we saw the presence of a single device failure may render the key exchange algorithm useless. Thus we need an approach where two devices can establish a new key even if there are device failures in the network. Furthermore, in order to keep the connectivity of the underlying key graph we need an approach for removing a device from the network even if this devices does not perform the procedure for controlled shutdown.

In order to make our key distribution scheme resilient against *Trudy* we need to modify our algorithms in a way that the above sketched attacks aren't possible.

Due to the numerous possibilities of a *Mallory*-type attacker, it is impossible with the presented class of algorithms to cope with every occurring situation. Furthermore, an attacker which is interested in hindering the normal operation of the network could attack more easily on the physical layer, e.g. by jamming the frequencies. Therefore, making the key distribution schemes resilient against *Mallory* involves countermeasures on all layers, especially the physical layer. Coping with *Mallory* is part of future work.

Therefore, in the next section of this chapter we present how to extend our basic approach such that the s-connectivity of the key graph is guaranteed even in the presence of device failures. Finally we provide in Section 4.3 the modifications in order to cope also with *Trudy* attackers.

4.2 Device Failures

In this section we extend our basic approach to cope with device failures. As we saw in the previous section the key establishment algorithm cannot guarantee the successful key establishment in case there are less than s device-disjoint paths available due to a device failure. Hence we introduce fault tolerance in Section 4.2.1 with the aid of path redundancy. Based on this path redundancy we then present in Section 4.2.2 an algorithm for recovery from *single* device failures. The algorithm for recovery sustains the connectivity of the key graph after such a failure. Based on this recovery algorithm we develop in Section 4.2.3 a second algorithm, which recovers the underlying key graph even in the presence of *multiple* device failures. In

Section 4.2.4 we give an extension in order to cope with devices which are temporarily suspended and should not be removed from the network. Finally, we conclude our approach for device failures in Section 4.2.5 by discussing the properties of all presented approaches.

4.2.1 Path Redundancy

An approach for dealing with device failures is the usage of redundancy. In our case we introduce path redundancy into the network, i.e. we provide more than s device-disjoint paths in order to have tolerant paths available. In the following, we first adapt our key graph construction algorithm in order to provide this tolerance. Then we present the adapted key exchange algorithm which uses the newly introduced redundancy. After that, we shortly discuss an approach for counteracting problems occurring due to device failures in the neighborhood graph G_N. Finally, we conclude this section with a summary of the properties of the newly introduced algorithms.

Key Graph Construction: Adding a new Device

In general, device failures lead to a key graph which is x-connected, with $x \leq s$. However, the algorithm for key establishment (see Procedure 3.4 and 3.5 in Section 3.3) needs an underlying key graph which is at least s-connected. To provide such a graph in the presence of device failures we need to build a key graph which is z-connected, with $z > s$.

The question which directly arises is what value z needs to have in order to cope with r device failures, while r is the *redundancy level* as defined in Section 2.4.

From a formal point of view, whenever a device fails its corresponding node and all associated edges are removed from the key graph. In order to determine the value of z we make the following observation: The removal of nodes can decrease the connectivity of the key graph by at most the number of removed nodes. Using Definition 3.1 of a k-connected graph from graph theory we can state:

Theorem 4.1 *In a k-connected graph $G = (V,E)$, the removal of t nodes (and all corresponding edges), will result in a graph, which will be at least $(k-t)$-connected.*

PROOF By contradiction. Assume that the graph G' induced by $V - Y$ with $Y \subseteq V$ and $|Y| = t$ is *not* at least $(k-t)$-connected. Then it suffices to remove h nodes from G', with $h < (k-t)$ in order to disconnect G'. But this means we disconnected G by removing $t+h$ nodes with $t+h < k$, which is a contradiction to G being k-connected since the removal of less than k nodes cannot disconnect G. ■

With this observation the value of z is easy to be determined: In order to cope with *at most* r device failures we need to establish a $z := (s+r)$-connected key graph. The adapted algorithm for our key graph construction is given in Algorithm 4.1.

Algorithm 4.1 Fault tolerant key graph construction – adding a node

1: Given a graph $G_K = (V, E_K)$ with $n = |V|$ with nodes $v_i \in V$ and a new node v_{n+1}
2: $V := V \cup \{v_{n+1}\}$;
3: **if** $(s+r) \geq n$ **then**
4: **for** $i = 1$ to n **do**
5: $E_K = E_K \cup \{v_{n+1}, v_i\}$; // provide a new pairwise key securely
6: **end for**
7: **else**
8: $V' :=$ random subset of $V - \{v_{n+1}\}$ with $|V'| = (s+r)$;
9: **for** $i = 1$ to $(s+r)$ **do**
10: $E_K = E_K \cup \{v_{n+1}, v_i\}$ with $v_i \in V'$; // provide a new pairwise key securely
11: **end for**
12: **end if**

The only difference in Algorithm 4.1 with respect to Algorithm 3.1 are lines 3, 8 and 9. Here we replace s with $(s+r)$. Thus, since the original algorithm constructed an s-connected graph, the modified algorithm constructs an $(s+r)$-connected key graph.

As already mentioned in the analysis part, this algorithm works also in the presence of device failures. In case one or more of the $(s+r)$ devices which have been chosen for introducing a new device into the network have failed, we simply need to choose other devices which have not failed. Hence device failures do not pose a problem for Algorithm 4.1. Note that choosing devices and detecting failures is done out-of-band, e.g. by the user.

If we apply Algorithm 4.1 directly in our different bootstrapping methods from Section 3.4 an $(s+r)$-connected key graph is constructed from the beginning. However, it is not necessary that the key graph construction algorithm constructs an $(s+r)$ key graph from the beginning. An alternative approach is to create an s-connected key graph with the key graph construction algorithm (i.e. using the original Algorithm 3.1) and establish the other r keys with the fault tolerant key establishment algorithm as described in the next paragraph. This is possible since our fault tolerant key establishment algorithm needs only an s-connected key graph in order to function properly.

This approach is especially useful when using the physical contact bootstrapping method as described in Section 3.4.3. In that case, the user needs to initialize his new device only with s already existing devices instead of $(s+r)$. If r is high, this is of great relief for the user. The approach however has a drawback: If one of the s devices which have been used for introducing the new device into the network fails before the new device could establish the first

of the r additional keys, the establishment of any further keys will fail. Thus the device cannot be entered successfully into the network. This is a minor drawback, since the probability that a device fails exactly in this moment can be assumed to be very low.

Fault-Tolerant Key Establishment

Our modified key graph construction algorithm for adding devices to the network (Algorithm 4.1) constructs an $(s+r)$-connected key graph, i.e. there exist $(s+r)$ device-disjoint paths between any pair of devices. Due to these additional r device-disjoint paths the key establishment algorithm from Section 3.3 can be adapted in order to guarantee the key exchange even in the presence of at most r device failures. For the adaption of the original algorithm from Section 3.3 we need to consider that a failed device, being on a secure path during key establishment of two other devices, will not forward the key shares. Hence, two issues need to be solved: First we need to detect such situations and second we need an appropriate reaction, so that the key establishment algorithm can recover and finally successfully establish the key.

There are several possibilities for detecting if a certain secure path is operational or not. For instance we could acknowledge every single hop on the secure path, and in case it fails at a certain point a message could be sent back along the path, which announces the initiating device. In our approach we choose a simpler approach in which the key share is forwarded to the destination over the secure path, and when the destination receives a key share it acknowledges this with an unsecured acknowledgement. Sending an unsecured acknowledgement has the advantage that we do not need to send the acknowledgement back along the path, but can simply send it directly to the initiating device (with which it obviously does not share a key yet). When the initiating device receives such an acknowledgement, it can be certain that the key share was received. On the other hand if the acknowledgement was not received within a certain timeout interval, the initiating device can assume that there was a problem with the corresponding secure path. In this case it needs to resend the key share over a different secure path – of which at least r are available. In case there are more than r device failures in the network, it cannot be guaranteed anymore that the initiating device can find another path, therefore the key establishment cannot be guaranteed anymore. Note that sending an unsecured acknowledge does not give an *Eve*-type attacker any advantages. However, a *Trudy* or *Mallory*-type attacker could use the unsecured acknowledgement in order to attack the system. Attacks and countermeasures are described in Section 4.3.

One possible adaption of the algorithm from Section 3.3 is given in Procedures 4.1 and 4.2. An example is shown in Figure 4.4.

Whenever a device needs to establish a key to another device Procedure 4.1 is locally called with the device ID of the target device (*TargetID*). Just like in the original key establishment

```
1:  procedure FaultTolerantKeyEstablishment(TargetID);
2:  Paths := DiscoverNodeDisjointPaths(TargetID);
3:  j := 0;
4:  for i := 0 to s − 1 do
5:     KeyShare := random(KEY_LENGTH);
6:     EstablishmentKeyStore.UpdateKey(TargetID, KeyShare); success := false;
7:     while not success and i + j < s + r do
8:        NextHop := Paths[i + j][0];
9:        SecureSend(NextHop, KeyShareTransport, {MyDeviceID, KeyShare, Paths[i + j]});
10:       wait for "FTKeyShareTransportACK" from TargetID
11:       if FTKeyShareTransportACK received within timeout interval then
12:          success := true;
13:       else
14:          j := j + 1;
15:       end if
16:    end while
17: end for
18: if i = s − 1 ∧ success = true then
19:    return true;
20: else
21:    return false;
22: end if
```

Procedure 4.1: Initiating fault tolerant key establishment

algorithm from Section 3.3 the initiating device needs to discover the device-disjoint paths to the target device. The difference here is, that we can guarantee due to Algorithm 4.1 that this function returns at least $(s+r)$ paths and stores them in *Paths*. After discovering these paths we need to send over s of these discovered paths a unique key share to the target device. Hence we loop the following steps s times. The first step is the generation of a random key share in the length of the key. This key share is added to *EstablishmentKeyStore* using the *Update* method. As described in Section 2.5 this method XOR's the key share to the already existing key of a particular ID. After storing the key share locally, it needs to be sent over a secure path to the target device. The difference here with respect to the original Procedure 3.4 is that we need to ensure that the key share reached its destination. This is done by letting the target device acknowledge the reception (see also Procedure 4.2) of the key share. In case we don't receive such an acknowledgment within a certain timeout value, we assume that the key share did not reach the target device, hence the secure path is broken. Thus we choose the next available path to send that specific key share. This is repeated until either the key share was successfully received or there are no more paths left.

The procedure terminates if all key shares could be sent successfully or if there are no more paths available. In the latter case, there are more than r device failures in the network, thus the key establishment cannot be guaranteed anymore.

```
1:  procedure onFTKeyShareTransport(Sender, {InitiatorID, KeyShare, Path[]});
2:  RestPath := Path - { Path[0] }; // Delete the first element
3:  if |RestPath| > 0 then
4:      // we are just transporting the share, thus we need to forward
5:      SecureSend(RestPath[0], KeyShareTransport, InitiatorID, KeyShare, RestPath);
6:  else
7:      // we are the receiver of this share, thus store the share
8:      if not duplicate then
9:          EstablishmentKeyStore.Update(InitiatorID, KeyShare);
10:     end if
11:     send(InitiatorID, FTKeyShareTransportACK, { } );
12: end if
```

Procedure 4.2: Receiving key establishment (fault tolerant)

Procedure 4.2 shows the slightly modified procedure which is executed when receiving a message with a key share. For all intermediary devices on a secure path the behavior is unchanged with respect to the original Procedure 3.5. The only differences are in line 8 and line 11, which are executed only on the target device. First, the receiving device must check if such a key-share was already received. This can happen if for instance a message was delayed long enough so that the sending device did already resend the key-share. Secondly, in line 11, the receiving device needs to send back the acknowledgement message to the initiating device. In order to do so, the basic `send` procedure is used, which does not use encryption to deliver the message. As already mentioned above, encryption for the acknowledgement is not necessary to fend off an *Eve*-type attacker and to cope with device failures. However, as we will see in Section 4.3 when dealing with *Trudy*, this can be harmful and we have to further refine this approach.

Note that the algorithm for fault tolerant key establishment presented in Procedures 4.1 and 4.2 describes a synchronous approach, i.e. the sender always waits for the acknowledgement before proceeding. The synchronous approach was chosen only for presentation simplicity. Clearly, in order to improve performance the initiating device would send in parallel all key shares along with an ID of the path. After the timeout interval the sending device would check the received acknowledgements. All key shares which have not been acknowledged are then resent on different paths. Note that acknowledgements for key shares from the previous set arriving after the timeout must be ignored – this can happen since it is not possible to determine an exact value for the timeout (asynchronous network). This is repeated until all key shares are successfully sent or no more paths are available.

Hence, with the adaption of the key establishment algorithm we can guarantee the successful key establishment as long as there are less than r device failures in the network.

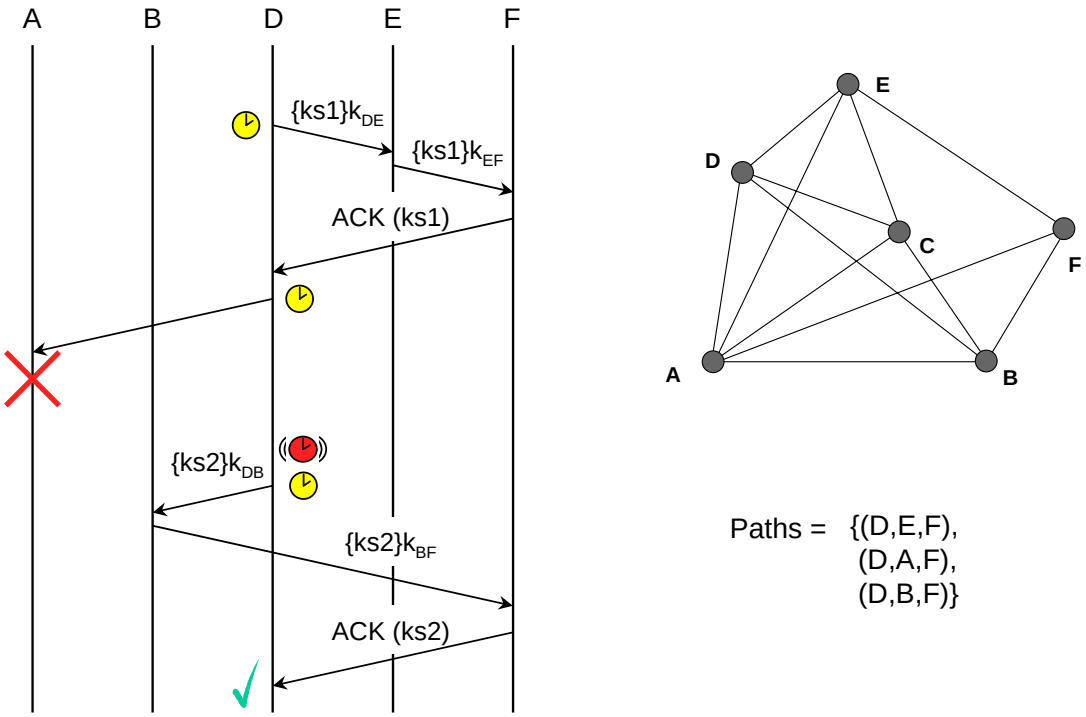


Figure 4.4: Fault-tolerant key establishment example ($s = 2$, $r = 1$)

As an example consider the key graph given in Figure 4.4. This key graph was constructed with a security level of $s = 2$ and a redundancy level of $r = 1$. In this example, device D needs to establish a new key with device F. To do so, device D executes a local call to the procedure given in Procedure 4.1 with device F as the argument. First, device D performs a discovery of at least three node-disjoint paths to node F. These paths are stored in the local *Paths*-variable and shown in Figure 4.4 in the lower right. Note that these are not all exiting paths, but only the three shortest.

After detecting the paths, device D creates a new key-share $ks1$ and sends it securely to device E which in turn will forward this key-share to device F. Whenever device D sends a message it also starts a timer in order to detect if the key-share did successfully reach it's destination. When device D receives the acknowledgement from device F for key-share $ks1$ it proceeds by sending the second key-share over a different path. The next path is (D,A,F), hence it sends the key-share $ks2$ securely to device A. However, shortly after receiving this message, device A crashes and does not forward the key-share. After some time, device D runs into the timeout. D deduces that the key-share could not be delivered over this path, and therefore chooses the next path to deliver the key-share. Hence, the key-share $ks2$ will be sent to device B, which in turn

forwards it to device F. After receiving the second acknowledgement, device D establishes a new key with device F.

Summary

As we have seen our fault tolerant key graph construction algorithm for adding new devices to the network constructs a $(s+r)$-connected key graph. Based on this algorithm our fault tolerant key establishment algorithm can guarantee the secure key establishment between arbitrary devices in the presence of at most r device failures. Each device failure reduces the connectivity of the key graph by at most one and thus if more than r devices have failed, the connectivity of the key graph may be lower than s. If a device gets deactivated and reactivated after some time (e.g. due to battery change), and all the keys are still stored on that device, it reduces the connectivity of the key graph only for the time being deactivated, i.e. after reactivating the device it is still part of the network. Hence, if e.g. the network administrator can ensure that at every point in time at most r devices have failed or are deactivated the successful key establishment can be guaranteed.

4.2.2 Recovery from Single Device Failures

In the last section, we showed that an $(s+r)$-connected graph can tolerate up to r device failures while retaining a security level of s. However, the connectivity of the graph will monotonously decrease with every additional failed device. We can enhance this with a recovery approaches that tries to sustain the network connectivity despite failures. The recovery can sustain connectivity if only up to a given number of failures occur simultaneously. More specifically, it assumes that the recovery process is finished before more failures occur. Clearly, this is not always the case. If more than the maximal number of simultaneous failures occur, the recovery approach can fail to sustain connectivity but will not lower the connectivity further.

As an example consider a sensor network with a lifetime of two years. During this time two devices fail, the first after six and the second after twelve months. Using path redundancy only, e.g. $s+2$, this leads to a network connectivity of s after one year. Therefore, no additional failures can be tolerated. If the network connectivity is recovered using a recovery approach, network connectivity will still be $s+2$, assuming that the recovery process did did not take more than six months.

In this section we extend the path redundancy by a recovery algorithm, which sustains the connectivity of the key graph after a single device has failed. In order to do so we modify our algorithm for controlled shutdown in a way that it can be executed also in case of a failure of a device. We present an extended version of this approach which recovers from multiple device failures in Section 4.2.3.

Before the system can recover from a device failure, it has to detect the failed device. Note that we assume fail-stop device failures, as described in our system model in Section 2.1. Thus, upon failure, a device stops communicating. Therefore, one way to detect a failed device is to detect that it is not answering requests anymore. Clearly, in an asynchronous system we cannot distinguish this from delayed messages. However, we can use a timeout mechanism to detect a failed device with high probability. If a device does not answer requests for a given time, we assume it to be failed and initiate recovery. This mechanism detects all failed devices (so called true positives) but may wrongly detect additional devices (so called false positives). In reality, given an adequate timeout value, the number of false positives will be very small. If a false positive occurs, the system defines the corresponding device as being failed, i.e., it ignores all future messages from this device. To include the device into the system again, it must be added completely anew into the key graph by performing the key pre-distribution algorithm (bootstrapping) again for this device. The same is true for failed devices that recover from their failure, e.g., because the user reboots them manually.

Based on this failure detection mechanism we now present two approaches for sustaining the key graph connectivity in the presence of single device failures. Our first approach is the straight forward adaption of the leaving algorithm from Section 3.2.2. The second approach optimizes the communication overhead with the introduction of data replication.

Gathering the Parents-set

When a device fails, it is not able to perform the procedure for controlled removal (Procedure 3.1 in Section 3.2.2). This procedure is based on the assumption that the device, which is to be removed from the network, has the time to announce its impending departure from the network and make all necessary arrangements. However, a failing device might not have this time. Therefore, to cope with silently failing devices, we need to adapt these algorithms.

The adapted removal algorithm is based on the controlled removal algorithm, but in contrast to it, it is initiated by a neighboring device instead of the leaving device itself. Note that by having an underlying $(s+r)$-connected graph, the removal algorithm can be executed without the aid of the device which has just failed. The basic idea behind the adapted removal algorithm is, that if some device in the network detects that a neighboring device has failed, it tries to recover this part of the underlying graph. In order to do so, it first has to gather the information which nodes are in the parents-set $\mathcal{P}$ (see Section 3.2.2) of the failed device. In the simplest case this information can be retrieved by broadcasting a request, to which all devices which are in the parents-set of the failed device answer. According to Algorithm 3.3 the detecting device now only needs to replace the key to the failed device to one device which answered the request and it does not share a key with yet. The adapted removal algorithm is executed in parallel on all devices which detect the failed device. Note that the broadcast procedure is insecure.

However, there is no need to secure these messages since an *Eve*-type attacker will not gain any information from such messages. As will be seen in Section 4.3, when analyzing active attackers this becomes important since such an attacker might use such messages to deliberately confuse the algorithm. In the following we give an example for the adapted removal algorithm before we describe the algorithm in detail.

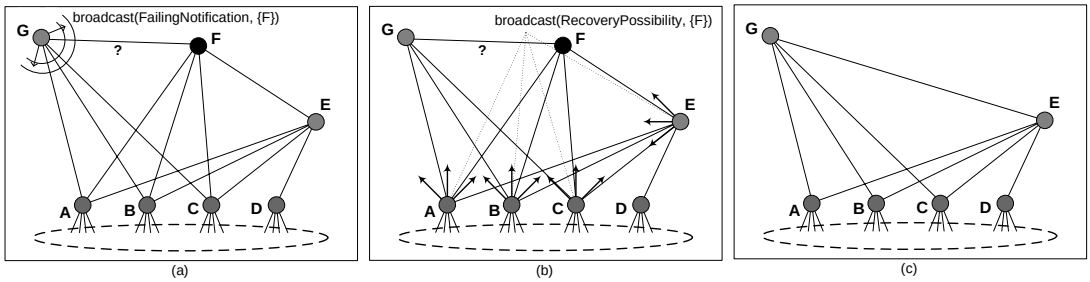


Figure 4.5: Recovery from single device failures ($s = 2, r = 2$)

As an example for our removal algorithm consider Figure 4.5. In this example we constructed a network with a security level $s = 2$ and a redundancy level $r = 2$, hence with a total connectivity of $z = 4$. The nodes in the parents-sets of devices A, B, C, and D are not shown in the figure – it is only important that A, B, C, and D where already part of the network when E, F, and finally G were added to the network. At some point in time device F fails. Device G detects the failure of F and broadcasts a "FailingNotification"-message (Figure 4.5(a)). Clearly, also devices A, B, C, and E might have detected the failure of F – for simplicity we omit the corresponding messages from Figure 4.5 and show the recovery process only from the viewpoint of device G. After devices A, B, C, and E receive the "FailingNotification" from device G they will respond by broadcasting a "RecoveryPossibility"-message (Figure 4.5(b)). They do so because device E was found in their children-set. Note that device D does not send such a message since the ID of device F was not found in its *KeyStore*. When device G receives a "RecoveryPossibility"-message it checks if it shares a key with the sender of the message. If this is not the case the sending device is a possible replacement for the failed device. In this example when G receives a "RecoveryPossibility"-message from device E it finds that it does not share a key with E. Hence it replaces the key to the failed device F with the new key to device E, using the key establishment algorithm (Figure 4.5(c)). After that, all keys referencing device F have been removed from all devices.

After giving an example for the algorithm we now describe it in more detail in Procedures 4.3, 4.4 and 4.5. Upon detection that a device has failed, the procedure `SingleRecovery` is called in order to signal this event (Procedure 4.3). This procedure is called with the device ID of the failed device as an argument. We use in this approach two further data structures, namely *RecoveringDevicesSet* and *RecoverySet*. Both data structures are sets, realized as lists. There are two methods available, namely *Add* and *Delete* which take both as an argument a

device ID which is to be added or removed from the set. The *RecoveringDevicesSet* is used in order to remember if a device is already in the process of recovering a certain failed device. That is, upon detection of a device failure, the corresponding ID of the device is added to the *RecoveringDevicesSet*. The *RecoverySet* is used in order to remember devices which have been in the children-set, even after the corresponding entry in the *KeyStore* has been removed. This is necessary, since upon failure of a device the corresponding entry in the *KeyStore* is deleted. However, later there might be some request for devices which had the failed device in their children-set. With the aid of *RecoverySet* such a request can be answered.

```
1:  procedure SingleRecovery(FailedDeviceID);
2:  if FailedDeviceID ∈ KeyStore.GetParentsSet() then
3:     // The device is part of the parents-set, thus we are in set C of the failed device.
4:     if FailedDeviceID ∉ RecoveringDevicesSet then
5:        RecoveringDevicesSet.Add(FailedDeviceID)
6:     end if
7:     broadcast(FailingNotification, {FailedDevice});
8:  else if FailedDeviceID ∈ KeyStore.GetChildrenSet() then
9:     // The device is part of the children-set, thus we are in set P of the failed device
10:    broadcast(RecoveryPossibility, { FailedDeviceID } );
11:    if FailedDeviceID ∉ RecoverySet then
12:       RecoverySet.Add(FailedDeviceID);
13:    end if
14:    KeyStore.DeleteKey(FailedDeviceID);
15: else if FailedDeviceID ∈ RecoverySet then
16:    // The device was part of the children-set but the key was already deleted
17:    broadcast(RecoveryPossibility, { FailedDeviceID } );
18: else
19:    // Nothing to to.
20: end if
```

Procedure 4.3: Recovery of a single device failure

Whenever a device A detects the failure of another device B it locally executes Procedure 4.3. It first checks if B is part of its own parents-set, i.e. if $v_B \in \mathcal{P}_{v_A}$. In case it is part of its parents-set, it needs to initiate the recovery since its corresponding node v_A from the key graph will be missing one of its original edges. First, the ID of the failed device is entered in the *RecoveringDevicesSet* in order to remember, that a recovery process is already underway. In order to replace the missing edge, the device needs to find a suitable replacement, that is, according to Algorithm 3.3, a device from the parents-set of the failed device. Even though the parents-set of the failed device is only known to the failed device itself (and therefore not available anymore), each device which has the failed device in its children-set must have been in the parents-set

of the failed device. Thus, A now broadcasts a "FailingNotification"-message in order to find other devices, which have the failed device in their children-set.

If the failed device B is in the children-set $\mathcal{C}_{v_A}$ of A there is no need for a local recovery. This is because all original edges of the node v_A from the underlying key graph are still existing. However, device A is therefore in the parents-set of the failed device, i.e. $v_A \in \mathcal{P}_{v_B}$. Thus device A might be a candidate for recovery for devices from the children-set of the failed device, i.e. $\mathcal{C}_{v_B}$. Therefore device A broadcasts a "RecoveryPossibility"-message. However, since the broadcast procedure is not reliable, this message might not be received by all devices which are in the children-set of the failed device. Therefore the ID of the failed device is added to the *RecoverySet*. As already mentioned, the *RecoverySet* is a set which contains all device IDs which have failed, and device A was in their parents-set. After adding the failed device to the *RecoverySet* the corresponding entry in the *KeyStore* can be deleted.

In case the failed device ID was found neither in the parents-set nor in the children-set, but in the *RecoverySet* the device needs to send a "RecoveryPossibility"-message in order to announce that this device was (since the key was already deleted earlier) in the parents-set of the failed device.

Finally, if the failed device ID was found in neither set, the call can be ignored, since the receiving device is not involved in the recovery process at all. This can happen since the `Recovery`-procedure is also called when receiving a "FailingNotification"-message as shown in Procedure 4.4.

Note that with this approach a lot of broadcasts might be sent. An alternative approach needing less broadcasts but more memory is presented in the next section.

```
1: procedure onReceiveFailingNotification(Source, {FailedDeviceID});
2: if FailedDeviceID ∉ RecoveringDevicesSet then
3:     SingleRecovery(FailedDeviceID);
4: end if
```

Procedure 4.4: Receiving a "FailingNotification"-message

When a device receives a "FailingNotification"-message it first checks if the recovery process for the failed device is not already running, i.e. if the failure was already detected either locally or through an earlier "FailingNotification"-message. It does so by checking if the ID of the failed device can be found in the *RecoveringDevicesSet*. If this is not the case, the `Recovery` procedure from above is called.

Finally, when a device receives a "RecoveryPossibility" from another device it first checks if the ID of the failed device is listed in the *RecoveringDevicesSet*, i.e. the failure was already detected and not yet recovered. If the device does not share a key yet with the sending device, a new key to the sending device will be established. After that, the key to the failed device is

```
1: procedure onReceiveRecoveryPossibility(Source,{ FailedDeviceID });
2: if FailedDeviceID ∈ RecoveringDevicesSet and Source ∉ KeyStore.GetParentSet() then
3:    // establish a new shared key with Source;
4:    FaultTolerantKeyEstablishment(Source);
5:    // move the key into the main key store
6:    KeyStore.Replace(FailedDeviceID, Source, EstablishmentKeyStore.GetKey(Source));
7:    EstablishmentKeyStore.DeleteKey(Source);
8:    // announce the other device, that it has a new child
9:    SecureSend(Source, NewChild, { });
10:   RecoveringDevicesSet.Delete(FailedDeviceID)
11: end if
```

Procedure 4.5: Completing recovery

replaced by the key established with the sender of the ”RecoveryPossibility”-message. Then the ID of the failed device must be removed from the *RecoveringDevicesSet* – otherwise, the failed device would be recovered multiple times, i.e. with each received ”RecoveryPossibility”-message.

As can be seen in the above procedures whenever a device A fails all devices which have the failed devices in their children-set, i.e. all devices from $\mathcal{P}_{v_A}$, need to store the ID of the failed device in the *RecoverySet*. The question that remains to be discussed is when to remove a device ID from the *RecoverySet*.

Removing the device ID too early could result in the failure of the recovery algorithm. A device which sends the ”FailingNotification”-message can only get an correct answer if the devices from the parents-set of the failed device still have knowledge about the failed device. Hence, if the ID is removed prematurely, devices from the parents-set of the failed device will not answer to the ”FailingNotification”-message and therefore take away any chance for the device which has sent the ”FailingNotification”-message.

Thus, the device ID needs to be available as long as the recovery process is running on one of the devices from the children-set of the failed device, i.e. $\mathcal{C}_{v_A}$. However, as we assume an asynchronous system, we cannot determine this time deterministically. In reality, we can use a timeout mechanism with a very large timeout, e.g. one day. As an alternative we can rely on the user to notify us once the recovery is finished.

Note that as long as a certain device ID has not yet been removed from all *RecoverySet*s it cannot be reused in the network e.g. when introducing a new device. Otherwise, the failure of this new device could result in an erroneous correction of the key graph. Therefore, a failed device that was removed from the network and is added again must – in general – get a new device ID.

To see why these algorithms are correct, consider the following: In Algorithm 3.3 we deduced that on failure of a node v_j any node $v_i \in \mathcal{C}_{v_j}$ has to replace its missing edge to v_j with an edge to a node $v_k \in \mathcal{P}_{v_j}$. In the above algorithms, a device will detect the failure and will therefore send a "FailingNotification". All nodes $v_i \in \mathcal{P}_{v_j}$ will receive and answer to this request with a "RecoveryPossibility", thereby giving all nodes $v_k \in \mathcal{C}_{v_j}$ the information needed to replace the missing edge.

As the above algorithms are based on the controlled removal algorithm, they provide the same properties, i.e. after a single device failure the key graph will eventually have the same connectivity as before.

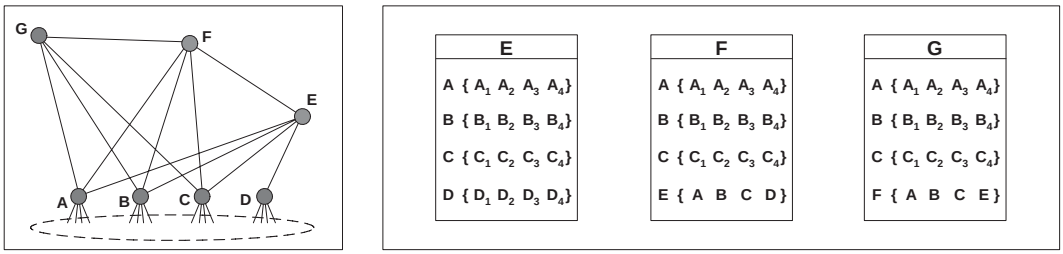
Replication of the Parents-set on all Devices from the Children-set

As already discussed, devices from the children-set of a failed device need the information about the parents-set of the failed device. Hence in the above presented recovery algorithm for single device failures the parents-set is gathered with broadcasts. However, broadcasts might stress the communication network. We can avoid these broadcasts if the devices from the children-set of a failed device have the knowledge about the parents-set somehow a priori. Hence the question arises, how to get the parents-set of a failed device without broadcasting.

In order to give an answer to that question we first make an observation. The parents-set of a device may change only if a device from the parents-set failed or left the network. Hence for a network, where device failures and removing of device is rare, the parents-set changes very rarely after the device has been introduced into the network.

With this observation in mind we developed an approach where we replicate the knowledge about the parents-set on exactly those devices which need this information in case of a failure, namely all devices from the children-set of a device.

As an example consider Figure 4.6 – this is the same network as was used in the previous example with $s = 2$ and $r = 2$. Again the nodes in the parents-sets of devices A, B, C, and D are not shown. Let the parents-set of device A be $\{A_1, A_2, A_3, A_4\}$, and analogously for B, C, D (see Figure 4.6(a)). We need to replicate the parents-set of e.g. device A on device G, F, and E, since these are the devices which would miss one of their original edges in case device A would fail. Analogously the parents-sets of the devices B, C, D, E, and F need to be replicated on all devices of their respective children-sets. Figure 4.6(b) shows the information stored on nodes E, F, and G. For instance, node E is in the children-set of devices A, B, C, and D, therefore it stores the parents-sets for these nodes (first column). The parents-set of device E is $\{A, B, C, D\}$. This information is replicated on device F since F is in the set $\mathcal{C}_{v_E}$ of node E (column 2, line 4). Similarly the parents-set of device F is $\{A, B, C, E\}$ which is stored only on device G (column 3, line 4). The parents-set of device G is $\{A, B, C, F\}$ which is not replicated



(a) Topology (b) Recovery Information

Figure 4.6: Recovery Information

since device *G* has an empty children-set. The replication of the parents-set of *G* is not needed, since in case that device *G* will fail no other device would miss one of its original edges.

In order to store the replicated parents-set on each device we introduce a new data structure, namely *ParentsReplicas*. There are three methods available. With the method *Add* we can add a new parents-set to the data structure. *Add* takes a device ID as the first argument and the corresponding parents-set as the second argument. In case there is already an entry present with the same device ID, the entry is overwritten with the new one. The method *Get* retrieves the parents-set from the local data structure by giving the device ID as the single argument. Finally, there is a method which allows deleting an entry from the data structure. This method takes as its single argument the device ID of the parents-set which needs to be removed.

The replication of the parents-set is done during the initialization phase of a device. Whenever a new device is introduced into the network, the initial keys are shared with other devices. Immediately after these keys are shared, the device has the knowledge about its own parents-set. Hence whenever another new device is added to the children-set of this device it sends to that device the list of IDs of its own parents-set using the message type ”ReplicateP” and its parents-set as the payload of the message. The device receiving this information needs to store it in *ParentsReplicas* data structure where it can be accessed later in case of a device failure (Procedure 4.6).

```
1: procedure onReceiveReplicateP(Source, ParentsSet);
2: ParentsReplicas.Add(Source, ParentsSet);
```

Procedure 4.6: Replicating the parents-set

As already mentioned, the parents-set may change during the lifetime of a device in case it was involved in the recovery process of another failed or leaving device. Clearly, in such cases also the replicated parents-set on all devices from the children-set need to be updated using the ”ReplicateP”-message.

With the aid of this replicated data and the new data structure *ParentsReplicas* we present in Procedure 4.7 the new `Recovery` procedure which can recover from single device failures. As before, this procedure is called upon detection that a certain device has failed.

```
1:  procedure ReplicatedRecovery(FailedDeviceID);
2:  if varFailedDeviceID ∈ KeyStore.GetParentsSet() then
3:     FailedDeviceParentsSet := ParentsReplicas.Get(FailedDeviceID);
4:     Candidates := FailedDeviceParentsSet − KeyStore.GetAllIDs();
5:     if Candidates ≠ ∅ then
6:        Device := pick randomly one from Candidates;
7:        establish a new shared key with Device;
8:        KeyStore.Replace(FailedDeviceID,Device,key);
9:        SecureSend(Device, NewChild, { });
10:       for all Child ∈ KeyStore.GetChildrenSet() do
11:          SecureSend(Child, ReplicateP, KeyStore.GetParentsSet());
12:       end for
13:    end if
14: else
15:    // FailedDeviceID ∈ KeyStore.GetChildrenSet()
16:    KeyStore.DeleteKey(FailedDeviceID);
17: end if
```

Procedure 4.7: Executed upon detecting a device failure

In Procedure 4.7 the device first checks if the failed device is part of its own parents-set. In case it's not part of the parents-set and thus part of the children-set, we can simply remove the key from the *KeyStore*. Note that there is no need to remember that we have been in the parents-set of the failed device as with the recovery procedure before since every device performs the recovery purely based on local knowledge. If the failed device is from our parents-set we can retrieve locally with the help of *ParentsReplicas* the parents-set of the failed device. With this information the procedure continues as before – we choose the right candidate, establish a new key with this device, store the new key in the main *KeyStore* and announce the other device, that we have just become a child of it. Finally, we need to update the own parents-set at call devices from our children-set by sending them the "ReplicateP"-message.

As already mentioned, whenever the parents-set of a device changes or a new client was added the information about the parents-set needs to be communicated. Hence we need to adapt the procedure for processing the "NewChild"-message as can be seen in Procedure 4.8

The only difference in Procedure 4.8 with respect to the original one (Procedure 3.3 on page 54) is in line 6. Here, after moving the key from the *EstablishmentKeyStore* to the main *KeyStore*, we need to send the parents-set to the newly added child.

```
1: procedure onReceiveNewChild(Source,{ });
2: // update key status
3: KeyStore.AddKey(Source, EstablishmentKeyStore.GetKey(Source));
4: EstablishmentKeyStore.DeleteKey(Source);
5: // replicate the parents-set at the new child
6: SecureSend(Source, ReplicateP, KeyStore.GetParentsSet());
```

Procedure 4.8: Moving a key to the main key store

Note that by executing the `ReplicatedRecovery` procedure as described in Procedure 4.7 on every device, there is no need for broadcasts anymore. Procedure 4.7 copes with single device failures and therefore fully replaces the procedure Procedure 4.3 from page 82. However, the controlled removal algorithm as presented in Section 3.2.2 can still be used as an optimization.

Our modified approach uses additional memory on each device. For a $z := s + r$-connected graph, we need on average $2z \cdot s_k$ bytes memory storage for the keys on each device, where s_k is the size of a key in bytes. On top of that, the following amount of storage is needed for the parents-sets: Each set contains z device IDs. If s_i is the size of a device ID in bytes, we need $z \cdot s_i$ bytes for a single set, of which z sets are stored per device. Thus the additional memory requirements for storing the replicated lists are $z^2 \cdot s_i$ bytes per device. The total required memory per device is $2z \cdot s_k + z^2 \cdot s_i$ bytes, which is not dependent on the network size n. This means that it scales with respect to the network size n, but the memory overhead with respect to the security parameters is $O((s+r)^2)$.

The reasons why this algorithm can recover from single non-simultaneous failures remain the same as in the previous section. The problem of multiple device failures, however, is still unsolved. Procedure 4.7 cannot recover when multiple devices simultaneously. The reasons are the same as with the broadcast-based removal approach (Procedure 4.3).

Summary

Both presented algorithms recover from single device failures, i.e. they eventually restore the connectivity of the underlying key graph G_K. In order to cope with one device failure at a certain point in time, i.e. simultaneously, we need to construct an $(s+1)$-connected key graph using Algorithm 4.1. Due to Theorem 4.1 the key graph will stay at least s-connected if a device fails. Therefore new keys can still be established and all key shared with the failed device can be replaced. After this recovery process is finished the key graph will be $(s+1)$-connected again. At this time another device may fail.

4.2.3 Recovery from Multiple Device Failures

In the last section, we presented an algorithm which can recover the connectivity of the key graph after a single device failure. In this section we extend this algorithm in order to cope with multiple device failures. We start by presenting an example of multiple device failures. We then analyze why the above presented algorithm for recovery does not cope with multiple failures. Based on that analysis we develop a new approach for recovery which can recover the connectivity of the key graph in the presence of multiple device failures. We conclude this section with properties of the new recovery approach.

Example of a Multiple Device Failure

In Figure 4.7 we present an example for multiple device failing. In this figure we show the same network as we already know from previous examples with $s = 2$ and $r = 2$. As we saw before, the single device failure of device F was successfully recovered with the recovery algorithm from Section 4.2.2.

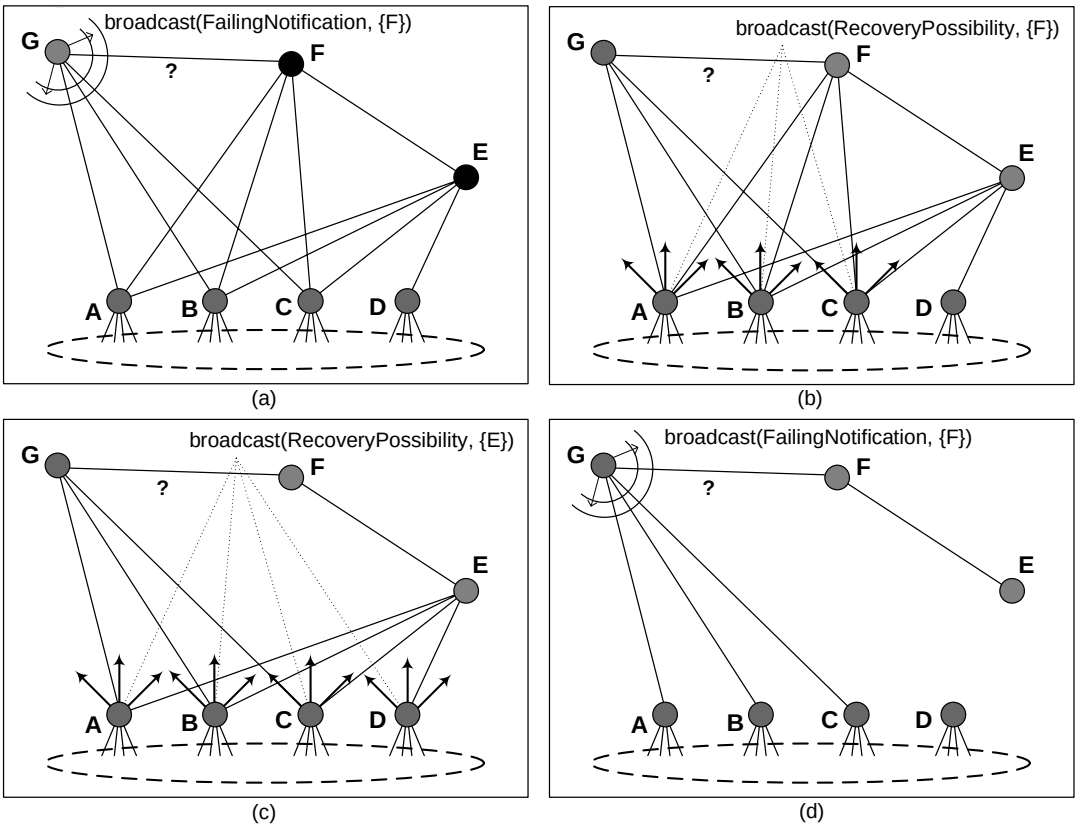


Figure 4.7: Failure to recover from multiple device failures ($s = 2$, $r = 2$)

Now, imagine that devices E and F failed at exactly the same time. Again, device G detects the failure of device F and using the broadcast approach (first presented method) from the last section it would broadcast a "FailingNotification"-message (Figure 4.7(a)). Clearly, the failure of E and F will also be detected by other devices, but for simplicity we concentrate on the recovery process from the viewpoint of device G. When devices A, B, and C receive the "FailingNotification" of device G they will correctly act according to Procedure 4.4 and 4.3 by broadcasting a "RecoveryPossibility"-message (Figure 4.7(a)) and deleting their own keys to device F. However, device E, which is also a device from the parents-set of device F, will not act at all since it failed, too.

When device G receives the "RecoveryPossibility"-messages it is not able to replace its key to F since it shares already a key with all devices which send a "RecoveryPossibility"-message.

Additionally the failure of device E was detected by devices A, B, C, and D which correctly broadcast a "RecoveryPossibility"-message for device E before removing their keys to device E (Figure 4.7(c)). Hence, after that step, every device but G removed their keys to both failed devices E and F. Since G was not able to recover its missing key to F yet, it will rebroadcast the "FailingNotification"-message (Figure 4.7(c)). Obviously, there will be no usable answer neither this time, since the only usable answer could come from device E which also failed. Thus device G will not be able to recover from the failure of the two devices E and F.

These multiple device failures also cannot be recovered by using the replication approach from last section. With the replication approach (Procedure 4.7) device G would try – upon detecting the failure of F – to establish a new key to device E. However, due to the failure of device E this key establishment cannot be successful. Hence the multiple device failures cannot be recovered.

Analysis of Multiple Device Failures

Let the node $v_x \in \mathcal{P}_{v_i}$ be a node corresponding to a failed device (i.e. $v_i \in C_{v_x}$). Hence, all edges (v_x, v_j) for any node v_j will be removed from the key graph G_K and thus v_i will miss one of its original edges. In Section 3.2.2 we showed, that such an edge needs to be replaced with an edge to some node from the parents-set of v_x, i.e. $\mathcal{P}_{v_x}$ (see Figure 4.8). Lets now assume, that there is a node $v_y \in \mathcal{P}_{v_x}$ of which the corresponding device also failed and thus also all edges from v_y were removed from the key graph G_K. Furthermore, let $\mathcal{P}_{v_i} - \mathcal{P}_{v_x} = \{v_x\}$ and $\mathcal{P}_{v_x} - \mathcal{P}_{v_i} = \{v_y\}$. Hence, the missing edge (v_i, v_x) needs to be replaced by (v_i, v_y). However, this is not possible since also node v_y was removed from the key graph. In Section 3.2.2 we analyzed that we need to replace a missing edge with an edge to an older node – the parents-set contains only older nodes. However, the only node v_y which could be used from the parents-set of v_x is also not available, thus we need to find another older node than v_x. Fortunately the parents-sets are transitive, i.e. any node $v_y \in \mathcal{P}_{v_x}$ is older than v_x and any node $v_z \in \mathcal{P}_{v_y}$ is older than v_y. Hence, any node $v_z \in \mathcal{P}_{v_y}$ is older than v_x, and therefore a suitable replacement for v_x.

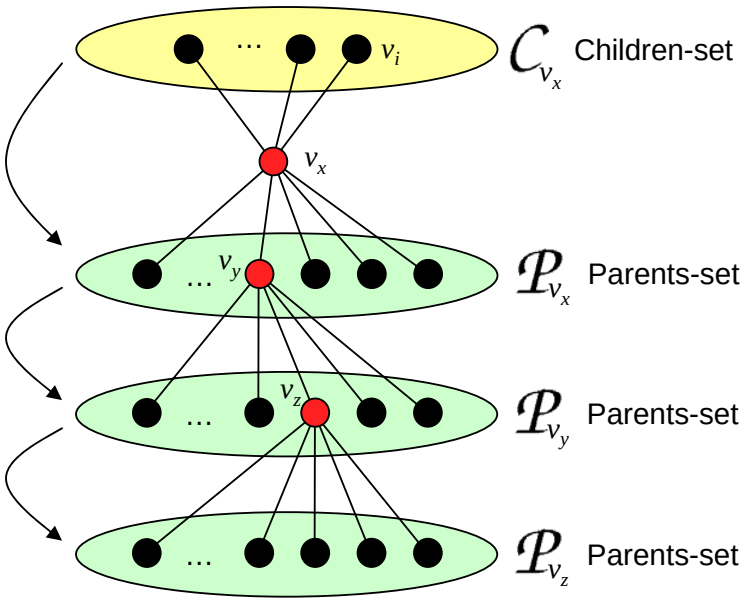


Figure 4.8: Transitive parents-sets

Informally this means, if the only suitable device Y in the parents-set of a failed device X has also failed, we can find a suitable device in the parents-set $\mathcal{P}_y$ of device Y. Clearly, this approach is not limited to two device failures. We just search in the parents-set of the oldest failed device. However, to identify this parents-set we have to be able to reconstruct a complete chain along the parents-sets of the failed devices up to the oldest one. Otherwise, the information which node's parents-set must be used could be lost. The algorithms used for single device failures cannot be used in this scenario as they are not able to reconstruct this chain in any case. Therefore, an additional algorithm is needed.

Replication of the Parents-set on all Neighboring Devices

Our algorithm for handling multiple failures works as follows. The parents-set of a node v_x is replicated on all nodes from its children-set $\mathcal{C}_x$ *and* its parents-set $\mathcal{P}_x$. Similar to the broadcast-based approach for handling single failures, the replication is done after a device has been introduced into the network and whenever the parents-set of a device changes due to a device failure. If a device $I \in \mathcal{C}_x$ detects the failure of X, it first examines the parents-set of X which is replicated on it. Hence it can locally determine a suitable device to replace the key to X. In case the only suitable device Y to replace X has also failed, I broadcasts a request for the parents-set of Y. Since the parents-set of a device is replicated on all $(s+r)$ parents of Y, at least one device will respond by sending the parents-set even if r devices have failed. With the information from the parents-set of Y, I can select another suitable device for replacing X and can try to establish

a new key with the chosen device. If this fails again, the device simply repeats the steps in order to gather the parents-set of the failed device. This is repeated until a suitable device for replacing the key to X is found. Procedures 4.9 to 4.12 describe this algorithm in detail.

```
1:  procedure MultipleRecovery( FailedID );
2:  if FailedID ∈ KeyStore.GetParentsSet() then
3:     if FailedID ∉ RecoveringDevicesSet then
4:        RecoveringDevicesSet.Add( FailedID );
5:        FailedParentsSet := ParentsReplicas.Get( FailedID );
6:        RecoverParent(FailedID, FailedID, FailedParentsSet);
7:     end if
8:  else if FailedID ∈ KeyStore.GetChildrenSet() then
9:     KeyStore.DeleteKey( FailedID );
10: end if
```

Procedure 4.9: Recovery of a failed device

Whenever the failure of a device is detected, the procedure `MultipleRecovery` is called, which is given in Procedure 4.10. We first distinguish if the failed device belongs to the parents-set or the children-set. In case it belongs to the children-set, we can simply remove the key to the failed device. Note that the replicated parents-set from the failed device is still stored in the *ParentsReplicas* data structure. This replicated data needs to be deleted. However, at this point in the algorithm it could still be needed and can therefore not be deleted yet – we discuss removing the replicate data after describing the rest of this algorithm. The more complex case is, if the failed device is part of the parents-set. First, we make sure, that we process this failure only once. Hence, we proceed only if we didn't find the ID of the failed device in the *RecoveringDevicesSet*. When the ID was not entered yet, we add it now to this data structure, thus locking out that this gets executed a second time.

In the next step we locally determine the parents-set of the failed device – this is possible since we replicated the information about the parents-set from all neighboring devices. With this parents-set as one argument we call the procedure `RecoverParent` which performs the algorithm for recovery in case the failed device is in the parents-set. This procedure is shown in Procedure 4.10.

The procedure `RecoverParent` takes three arguments: the ID of the failed device, the ID of an linked device which also failed and the parents-set of the linked device. In case it is called for the first time, there is no linked device failure, hence it is called from Procedure 4.9 with the ID of the failed device as the first and second argument and the parents-set of the failed device as the third argument.

In Procedure 4.10 we first calculate the set of devices which could replace the originally failed device. Here, the variable *Candidates* is filled with all device IDs which are in the provided

```
1:  procedure RecoverParent(FailedID, LinkedID, LinkedParentsSet);
2:  Candidates := LinkedParentsSet − KeyStore.GetAllIDs();
3:  for all ID ∈ Candidates do
4:    if FaultTolerantKeyEstablishment(ID) then
5:      KeyStore.Replace(FailedID, ID, EstablishmentKeyStore.GetKey( ID ));
6:      EstablishmentKeyStore.DeleteKey( ID );
7:      ParentsReplicas.Delete( FailedID );
8:      RecoveringDevicesSet.Delete( FailedID );
9:      RecoveringDevicesSet.Delete( LinkedID );
10:     SecureSend(ID, NewChild, { });
11:     for all Neighbor ∈ KeyStore.GetAllIDs() do
12:       SecureSend(Neighbor, ReplicateP, KeyStore.GetParentsSet());
13:     end for
14:     return
15:   end if
16: end for
17: LinkedFailure := pick one randomly from Candidates
18: broadcast(GetParents, { FailedID, LinkedFailure });
19: RecoveringDevicesSet.Delete( LinkedID );
```

Procedure 4.10: Recovering a device from the parents-set

parents-set but not in the main key store, hence all devices from the parents-set to which there is no key yet.

In the next step we try to establish a key with a device ID from the *Candidates* set. If this key establishment is successful we are almost done in recovering the device failure. In order to complete the recovery we just need to move the newly established key from the *EstablishmentKeyStore* to the main *KeyStore*, thus replacing the old entry to the failed device. After that we remove the failed IDs from the *RecoveringDevicesSet* and the replicated data for the failed device from *ParentsReplicas*. Then we notify the device to which a key was just established about the new circumstances. This new key belongs to the key graph, i.e. it needs to be moved into the main *KeyStore*. Finally, we have to re-replicate our parents-set on all neighbors since it changed and return from the `RecoverParent` procedure.

In case the key establishment was not successful, we try to establish a key to another device from the *Candidates*-set. This is done until we successfully established a new key or there are no devices left in the *Candidates*-set.

If we could not establish a key to any device from the *Candidates*-set, we have to recover from multiple device failures. In such a case, we pick one device from the *Candidates*-set and broadcast a request for the parents-set of this failed device using a ”GetParents”-message. Note

that if we reached that part of the `RecoverParent` procedure, it is clear that the device from the *Candidates*-set must also have failed – otherwise we would have been able to establish a new key to it in the steps before. The ”GetParents”-message includes the ID of the originally failed device (*FailedID*) and the ID of the device which we are currently trying to recover (*LinkedID*).

```
1: procedure onReceiveGetParents(Source, { FailedID, LinkedID });
2: if LinkedID ∈ ParentsReplicas then
3:     // a replica is available if LinkedID is or was part of the main key store
4:     LinkedParentsSet := ParentsReplicas.Get(LinkedID);
5:     send(Source, ParentsSet, { FailedID, LinkedID, LinkedParentsSet } );
6: end if
```

Procedure 4.11: Receiving a ”GetParents”-message

In Procedure 4.11 we show the reaction of a device which just received a ”GetParents”-message. First, the device checks if there is a replica of the searched parents-set available. If such a replica cannot be found, this device cannot help in this recovery process. When the parents-set for the failed linked device is found in the local *ParentReplicas* the device needs to send the parents-set to the requesting device using a ”ParentsSet”-message. In general this can be done by sending it directly and unencrypted to the requesting device. This is not an issue since it is not valuable information for an *Eve*-type attacker.

```
1: procedure onReceiveParentsSet(Source, { FailedID, LinkedID, LinkedParentsSet });
2: if FailedID ∈ RecoveringDevicesSet ∧ LinkedID ∉ RecoveringDevicesSet then
3:     RecoveringDevicesSet.Add( LinkedID );
4:     RecoverParent(FailedID, LinkedID, LinkedParentsSet);
5: end if
```

Procedure 4.12: Receiving a ”ParentsSet”-message

When a device receives a ”ParentsSet”-message, Procedure 4.12 is called. It first checks if this is a message for an ongoing recovery process and whether we don't process this already. This is done by checking if *FailedID* can be found in the *RecoveringDevicesSet*. If this is the case then this device is processing the failure of *FailedID*. Additionally if *LinkedID* cannot be found in *RecoveringDevicesSet* this is the first parents-set received for this *LinkedID*. Hence, if both checks succeed we add *LinkedID* to the *RecoveringDevicesSet* in order to not process this multiple times. Then we call again the `RecoverParent`-procedure with the received parents-set as one of the arguments. Hence the combination of `RecoverParent` and `onReceiveParentsSet` will loop until a suitable device ID can be found in order to replace the key to the first failed device.

After we saw how the algorithm works we can address the question when a replicated parents-set of a failed child can be deleted. The answer is similar to the answer we gave in the algorithm for recovering from single device failures with the information in the *RecoverySet*. According

to our algorithm the parents-set needs to be available as long as the recovery process is running on one of the devices from the children-set of the failed device, i.e. C_{v_A}.

Note that as long as a certain parents-set for a certain ID has not yet been removed from all *ParentReplicas* this ID cannot be reused in the network e.g. as a new ID when introducing a new device. Otherwise, the failure of this device might result in an erroneous correction of the key graph. On the other hand removing the parents-set to early, i.e. if there are still some devices which did not complete the recovery, would result in the failure of the recovery algorithm. A device which sends the "GetParents"-message can only get an correct answer if at least one device still possesses the knowledge about this parents-set. Hence, if it is removed prematurely, a device from the children-set of a failed device may not be able to determine a suitable device for replacing the key to the failed device and therefore take away any chance for the device which needs to recover the failure of a device from its own or linked devices parents-set. The best practice would be to never reuse a device ID, since we cannot guarantee at any point in time that the recovery process is already finished.

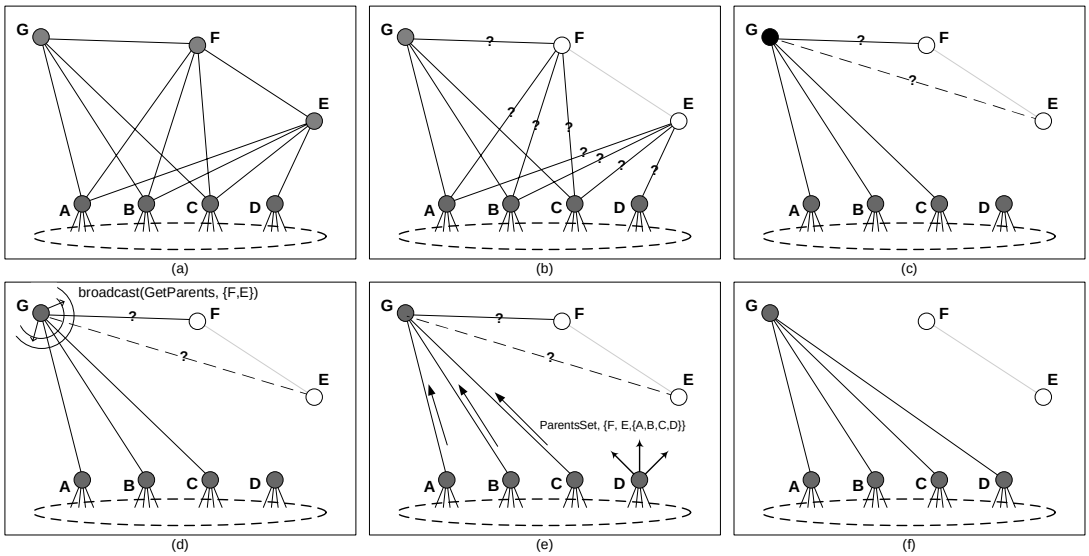


Figure 4.9: Recovery from multiple device failures, $(s = 2, r = 2)$

As an example we pick up the example from above with $s = 2$ and $r = 2$ depicted in Figure 4.9. Again we imagine that devices E and F have failed and all devices which share a key with these device detected that (Figure 4.9(b)).

According to Procedure 4.9 all devices which have the failed device in their children-set simply remove the key with these devices. In Figure 4.9(c) these are devices A, B, C and D. Device G however, has the failed device F in its parents-set and hence it executes Procedure 4.10. It will locally determine that the only candidate for replacing the key to F is device E and try to establish a new key to device D (the dotted line in Figure 4.9(c)).

Obviously the key establishment to device E will fail since device E also failed. Therefore device G needs to find another suitable device for replacing the key to F. It needs a device from the parents-set of device E, hence it broadcasts a request asking for this parents-set (Figure 4.9(d)).

All devices which receive this request, i.e. the "GetParents"-message, and have the necessary information stored in their *ParentsReplicas* will answer this request by sending a "ParentsSet"-message with the parents-set of device E (Figure 4.9(e)). Note that it would be sufficient if only the devices from the parents-set would respond to this request.

After device G has received the parents-set of device E according to Procedure 4.12 it executes the `RecoverParent` procedure. Here it determines that device D is the only suitable device for replacing the key to device F. After the key establishment to device D was successful, the recovery process is also for device G complete and the connectivity of the key graph has been restored to $z = 4$ (Figure 4.9(f)).

Our approach for multiple device failures is based on an $(s+r)$-connected key graph. Hence, it has the security of our basic approach, i.e. it tolerates at most $s-1$ *Eve*-type attackers. Furthermore, as long as there are at most r simultaneous device failures, the recovery algorithm can eventually reestablish the $(s+r)$-connectivity of the underlying key graph. After the recovery process has finished additional device may fail.

4.2.4 Temporarily Suspended Devices

In the last section we presented an approach which automatically removes a device from the network if it failed. Additionally we showed in Section 3.2.2 how to explicitly remove a device from the network (shutdown). In both cases, to add the device again to the network, a new device ID must be assigned and the bootstrapping process must be repeated. This is a rather cumbersome procedure and might not always be what we want. In certain scenarios it is preferable to keep an unreachable device in the network. This allows the device to resume its operation at once, after it becomes available again. As an example scenario, imagine a sensor and actuator network used for home automation. Some of the devices are battery powered. To change the batteries of a device, we suspend it, exchange the batteries and turn it back on. We expect that the device is still part of the network in the moment it is turned back on again. No further user interaction should be required.

To do so, we allow devices to be suspended and resumed, e.g. by manually pressing a button at the device. In order to do so, the device announces to all neighbors, i.e. all devices from the *KeyStore*, that it will not be available until further notice. We could do so by sending a "SUSPENDING"-message to all devices in the G_K-neighborhood. Each device receiving such a message, suspends the failure detection mechanism for this device. When the suspended

device is activated again it sends an ”ALIVE”-message to all neighbors in order to announce that it is back and operational. All devices will now re-activate failure detection for this device. However, if a device is deactivated, without removing it from the key graph, as shown in the previous section, it decreases the graph connectivity by one. Therefore if we build a $(s+r)$-connected key graph we can deactivate at most r devices. However, one must be aware that the number of deactivated devices directly influences the number of recoverable device failures, i.e. when deactivating r devices no device failures are tolerated.

In addition if we know already when introducing a new device into the network, that this device will become often deactivated we can choose another approach. We mark this device as a *bordernode* and define that if a device is marked as such it cannot act as a parent device for any other device. With other words, the children-set of a border node will always be empty. If the children-set of a node is empty, this node can be taken out of the key graph without any effect on the connectivity of the graph. Therefore, bordernodes may be suspended for unlimited time. Also the number of suspended bordernodes at the same time, is unlimited. If the failure detection is deactivated for bordernodes they can be suspended without prior announcement.

4.2.5 Summary

In this section we presented an approach for coping with sudden device failures, i.e. device which leave the network silently without executing the controlled shutdown algorithm as presented in Section 3.2.2. We first introduced the concept of path redundancy by establishing an $(s+r)$-connected graph. With this approach we can already tolerate up to r device failures. If more than r devices would fail, the successful establishment of new keys cannot be guaranteed anymore.

Based on the path redundancy we extended our approach with a recovery algorithm for single device failures. When a device fails the recovery algorithm re-establishes the original connectivity of the key graph. After this, another device may fail.

In addition to this, we also defined an approach for multiple simultaneous device failures. It extends the first recovery algorithm and allows to trace parent sets over multiple failed devices. Again, the recovery approach re-establishes the original connectivity of the key graph and allows additional devices to fail after this process is finished.

4.3 Active Attacker

Our basic approach can guarantee the secrecy of a newly established key as long as there are less than s devices subverted by an *Eve*-type attacker. In the last section, we extended our basic

approach in order to cope with device failures, also. As shown in Section 4.1, for *Trudy*- or *Mallory*-type attackers, the algorithms will fail and neither the secrecy nor the successful key establishment can be guaranteed anymore.

In this section, we describe the possible attacks of active attackers. First, we analyze how a *Trudy*-type attacker can attack the algorithms and give countermeasures for these attacks. Second, we discuss *Mallory*-type attackers. As already discussed, we do not provide resilience against such attackers in this work. Nevertheless, we analyze possible attacks to allow the reader to estimate how vulnerable our algorithms are towards a *Mallory*-type attacker. No countermeasures for the presented attacks are given.

4.3.1 *Trudy*-type Attackers

As already mentioned in our introducing analysis, a *Trudy* type attacker can mount two active attacks in order to intrude the network. In both cases, the participants would not notice the presence of *Trudy*, even though she gained access to new established keys. In the following we analyze in detail the two sketched attacks – i.e. the man-in-the-middle and the Sybil attack – and provide possible countermeasures.

Man-In-The-Middle Attack and Countermeasures

If *Trudy* wants to gain access to a new key between devices *A* and *B* she can launch a man-in-the-middle attack as shown in Figure 4.10. First, *Trudy* will discover *s* device-disjoint secure paths to device *A*. Now she generates *s* key shares and sends them over the *s* device-disjoint paths to *A* using the ”FTKeyShareTransport”-message. However, she will include the ID of device *B* as the *InitiatorID* (see Procedure 4.1 and 4.2). Note that no device is able to detect this deception. Device *A* receives these key shares and stores it in its own (secondary) key store but with the ID of device *B*. Furthermore, device *A* will send acknowledges to device *B* – but since device *B* is not waiting for any acknowledgement, these messages will simply be discarded.

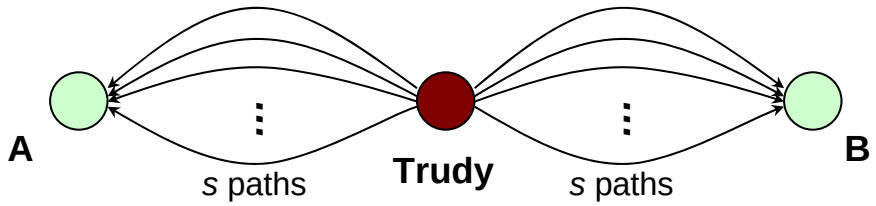


Figure 4.10: Man-in-the-middle attack

After *Trudy* established a new key with device *A* she does the same with device *B*. After that, she holds two keys (possibly the same) which can be used in order to impersonate device *A* against *B* and vice versa. If device *A* now sends a secure message to device *B* it will use the key from its establishment key store. If *Trudy* used two different keys for *A* and *B* she does not even need to intercept this message, since when it arrives at device *B* it will be discarded due to decryption failure. *Trudy*, however, can read the message - do whatever she wants to do with it, and then decide if she wants to forward it to *B* such that *B* can respond, so *A* does not become suspicions.

This attack is possible since although each message is authenticated per each hop, there is no authentication from end-to-end. Hence, when a device – let's say device *A* from the above example – establishes a key with some other device, it cannot be sure that it really establishes a new key with the ID which is pretended in the key share transport messages.

To prevent this attack, we need to integrate a classical challenge-response mechanism in our key exchange protocol. With such a mechanism both devices can be sure that they talk to the right device. In detail we modify the sending of the acknowledgement. When receiving a key share, the corresponding acknowledgement must be sent securely over device-disjoint paths. Whenever a device receives a key share it additionally discovers *s* device-disjoint secure paths to the claimed initiator. Over these paths the acknowledgment is sent. Additionally we need to piggyback a random number encrypted with the specific key share in each of these acknowledgment-parts (challenge). Note that due to our assumption *Trudy* has less than *s* devices subverted, she is not able to gain access to all random numbers. A device receiving these acknowledgments and really being the initiator of a key establishment, will collect all random numbers and perform some mathematical operation on them, i.e. simply add them. It will then send back the result encrypted with the newly established key. After that both devices are authenticated against each other – before that point in time the established key must be considered as invalid and must not be used.

With the additional measures as described above it is no longer possible for *Trudy* to mount a man-in-the-middle attack. Imagine she sends - just like in the example above – the manipulated key shares to device *A*, which in turn will send along *s* device-disjoint paths an acknowledgement with some nonces to device *B*. Device *B* however, knows that it did not initiate a key establishment with device *A*, so it will not confirm by sending the result. Hence, device *A* will not use the new key and will run eventually in a timeout and therefore remove the fake key. Furthermore there is no way for *Trudy* to send this confirmation since she lacks the information of at least one nonce (assuming she has at most $s-1$ devices subverted).

Note that the initiating device can be sure that it really talks to the intended device when it has received all acknowledgments for the key shares sent and it was able to decrypt each of them. Hence, by integrating this challenge-response mechanism we provide a two-way authentication

from end-to-end, which is resilient against *Trudy* as long as she was not able to subvert more than $s-1$ devices.

Sybil Attack and Countermeasures

To gain access to newly established keys *Trudy* must subvert at least s nodes in the key graph. One way to do so is to subvert s devices. Alternatively, *Trudy* could subvert one device and add it into the key graph multiple times resulting in multiple nodes representing the same device. Such an attack is called a Sybil-attack and is depicted in Figure 4.11.

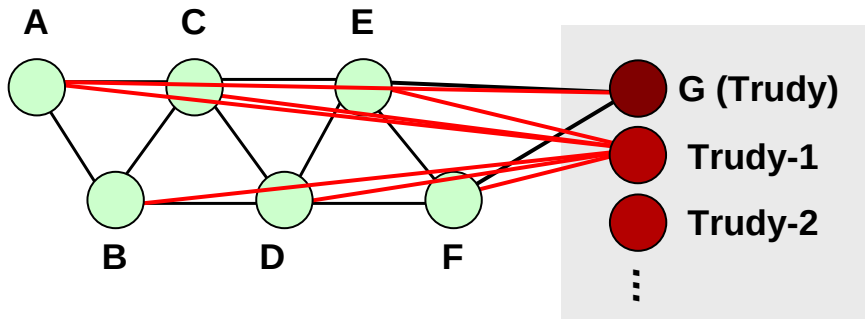


Figure 4.11: Sybil attack

In order to launch a Sybil-attack, *Trudy* just picks randomly s devices from the network and initiates a key establishment with them. However, she again fakes the identity - she claims that there is a device "behind" her – which actually does not exist but is only a virtual ID invented by *Trudy* - let's call it *Trudy*-1. In this case the key establishment will succeed - especially the countermeasures from above do not work, since here *Trudy* has full access (it is herself) over the initiator of the key establishment. Therefore, she can send challenges and respond correctly to them. Hence, she introduces a new virtual ID into the network with only one subverted device. Clearly, she could do that as many times as she needs in order to introduce any number of virtual IDs. With the aid of the "NewChild"-message, which is needed for the recovery algorithms, she can additionally make these keys part of the main key store, i.e. she is able to generate edges in the key graph. The more virtual IDs she introduces, the higher the likelihood that all s key shares of some new key establishment will actually be sent over her virtual IDs, i.e. through *Trudy* herself. Hence, she can access newly established keys with only one subverted device.

This problem arises since we assumed that the used paths are always device-disjoint – especially when receiving key shares. However, it is not enough, that one device discovers these paths and then initiates the key establishment. By doing that, we implicitly trusted the key establishment initiator. However, if the initiator is subverted by *Trudy*, it should not be trusted. Hence, all devices being part in the key establishment must check the disjointness of the paths.

Trudy was able to initiate key establishments, claiming it was initiated from devices which lie ”behind” her, i.e. these devices are only reachable through her – this is not allowed and must be prevented.

The first step to prevent this, is that the path along which a key share is sent, is not shortened at every device – only a pointer is moved on. Now, on each device the whole path is available, especially at the receiver of a key establishment. Before saving the key, the receiver must check if all key shares did really come over device-disjoint paths. With this modification, *Trudy's* attack from above is already not possible, since the contacted devices would recognize that the key shares from *Trudy*-1 have all been routed over *Trudy*. However, this is not enough since *Trudy* might simply write something different in the paths, in a way, that she no longer is a common device on all paths. This can be prevented, by letting every device additionally check its predecessor on the path, i.e. if the previous device was really the one written in the path. In case of a wrong path entry, the packet is simply not forwarded, hence the fake key establishment cannot succeed.

With these modifications *Trudy* is no longer able to introduce new virtual device IDs into the network. She still can establish keys at will with as many devices as possible, she even can add edges in the key graph. However, due to the s disjoint paths property of our key establishment she will never be able to be on all paths with less than s subverted devices.

4.3.2 *Mallory*-type Attackers

Due to the definition of *Mallory*, this attacker type is not bound to any limits. Similar to *Trudy*, she can arbitrarily send messages or decide not to send in situations where the protocol dictates that a message needs to be sent. However, *Mallory* and *Trudy* have different goals (see Section 2.1). *Trudy* aims at learning new keys. The goal of *Mallory* is to hinder the correct functionality of the protocol, i.e. to perform a denial-of-service attack. Just like the other attacker types, she can only encrypt or decrypt messages if she is in possession of the right key – which we assume if and only if she subverted the corresponding device from the network.

Clearly, a *Mallory*-type attacker could hinder the correct execution of all protocols in the basic approach from Chapter 3. In order to do so, she simply needs to deactivate a device, or hinder it to send any message. Hence, the device would behave like a device failure. Therefore we concentrate our analysis on the fault tolerant key establishment protocol and assume that we have an $(s+r)$-connected key graph as described in Section 4.2.1.

Besides behaving exactly like a device failure, a *Mallory*-type attacker has the possibility to send messages in any protocol. As already discussed the primary goal of a *Mallory*-type at-

tacker is to hinder the establishment of new keys. There are several ways to reach this goal. The most important ones are:

- The easiest approach to hinder the correct key establishment can be executed by *Mallory* only if she has subverted a device on the secure path between two devices which need to establish a new key. In this case she can directly interfere with the key establishment protocol.
- Besides directly interfering with the key establishment protocol, *Mallory* could try to decrease the connectivity of the key graph below s. If she succeeds the establishment of new keys cannot be guaranteed anymore.
- *Mallory* could flood a device with messages, in order to hider that device to correctly execute a protocol. With this approach she could directly interfere in the key establishment protocol or decrease the key graph connectivity.

In the following, we analyze these possibilities by looking at each protocol from the fault tolerant key establishment protocol.

Attacks on the Fault-Tolerant Key Establishment Algorithm

In order to establish a new key, a device chooses s paths (out of a total of $(s+r)$) to the destination and invokes the key establishment protocol as described in Section 4.2.1. If a device from the s chosen paths is subverted by a *Mallory*-type attacker, she has different possibilities in order to reach her goal, i.e. hindering the successful key establishment.

Mallory could simply decide not to forward the key share. However, by doing this she would not reach her goal. According to Procedure 4.1 from Section 4.2.1 the initiating device would simply chose another path after not receiving an acknowledgement. Hence, if she did not also subvert a device on this new path, *Mallory* would have locked herself out. Thus, this approach is of no use for *Mallory*.

A more fruitful approach for *Mallory* is to alter the forwarded key share. In that case, the protocol for fault tolerant key establishment would be executed correctly, but in the end, the two device would not share the same key. Hence, the key establishment was not successful and *Mallory* reached her goal.

Alternatively *Mallory* could attack the acknowledge messages which are sent unencrypted and also not authenticated. She could simply generate these acknowledgments by herself, and thus fooling the initiating device that everything is ok. In combination with not forwarding a key share, this would also lead to an unsuccessful key establishment.

Finally she could start by generating bogus key share messages – with an initiator ID which might not even exist – and sending them to a certain device. Since this device has no way of knowing that this is a faked key establishment it will store the received key shares in its *EstablishmentKeyStore*. Since the memory of our devices is limited, *Mallory* can fill up the memory of any device with unfinished key establishment, hence hindering any device from being able to participate in an authentic key establishment.

Clearly the fault tolerant key establishment protocol cannot guarantee the successful key establishment in the presence of a single *Mallory*-type attacker. Hence, in future research the above shown possibilities need to be eliminated in order to cope also with this attacker type.

Attacks on the Controlled Removal Algorithm

In Section 3.2.2 we introduced the algorithm for controlled shutdown of a device, i.e. a device leaving the network, which has the necessary time to make all arrangements in order to keep the key graph connectivity unaltered. The algorithm is given in Procedure 3.1 and 3.2. These procedures remain the same, also in case we have an $(s+r)$-connected key graph. According to these procedures whenever a device is removed in a controlled fashion from the network it is expected to execute Procedure 3.1. By interfering with this protocol *Mallory* does not directly interfere with the key establishment, but indirectly by trying to decrease the connectivity of the key graph – which in turn later could result in unsuccessful key establishments.

If *Mallory* subverted a device which is to be removed from the network her simplest way for interfering is by not executing the protocol for leaving. However, this might not be very fruitful, since from the viewpoint of the other device, the leaving device just failed. Hence, after some time the linked recovery protocol will kick in and restore the key graph connectivity.

Alternatively for the case that *Mallory* subverted a device which is to be removed from the network, she could provide a false parents-set along with the ”LeavingIntention”-message. According to Procedure 3.2 this parents-set would be used in order to correct the key graph hence by providing the wrong parents-set it is probable that the connectivity of the key graph has not been restored and thus *Mallory* would have successfully decreased the key graph connectivity. However, the parents-set supplied by the leaving device is actually not necessary anymore since we use in parallel the protocol for device failures where actually parents-set are replicated. Thus with a small change in Procedure 3.2, i.e. using the replicated parents-set instead of the provided one, this attack of *Mallory* would not accomplish her goal. Note that this only works under the assumption that the correct parents-set was replicated before *Mallory* subverted the device.

When a device receives a ”LeavingIntention” from a neighboring device, it must act according to Procedure 3.2 in order to repair the $(s+r)$-connected property of the graph. It must (if

necessary) establish new keys in order to bypass the leaving device and when done, send an acknowledgement to the leaving device.

In case a device which is subverted by a *Mallory*-type attacker receives such an message it might decide to not react at all. In that case the leaving device would not become the necessary acknowledgment, and after a certain timeout would leave the network anyway. If *Mallory* did not delete the key to the leaving device, at some point in time the recovery for device failures would kick in. Since *Mallory* wants to decrease the connectivity of the key graph, she will also prevent the recovery protocol from restoring the key graph connectivity. However, she is not as successful as she might think. As of Theorem 4.1 with this action she could achieve to decrease the connectivity of the key graph by at most one. Furthermore, this one path which is now missing between two other device in the network is clearly a path with the subverted device on it. Hence, *Mallory* actually limited her own chances of being on the secure path in order to directly interfere with a new key establishment.

Attacks on the Algorithm for Multiple Device Failures

In Section 4.2.3 we introduced an algorithm which recovers the key graph connectivity in the presence of r device failures. As we saw this algorithm is comprised of two steps: In the first step, single device failures are handled. If this is not successful the algorithm tries in the second step to recover from multiple device failures by requesting the parents-set of the failed device via a broadcast message. The goal of a *Mallory*-type attacker with respect to this protocol is to decrease the key graph connectivity in any situation possible, i.e. indirectly hindering the establishment of new keys by reducing the key graph connectivity below s. There are numerous ways for *Mallory* to do so. In the following we describe a few of them.

The simplest attack of *Mallory* is clearly not reacting at all if one device in the neighborhood fails. Since the attacker acts otherwise normal, especially responding correctly to all ”PING”-messages, the attacker will not be recognized as a failure, hence the protocol for multiple device failures will fail to recover the key graph connectivity. This might reduce the key graph connectivity by one with each device failure.

Even if the failed device is not in the direct key graph neighborhood of the device subverted by *Mallory*, she still can hinder the correct execution of the protocol. This can be done by providing a wrong parents-set in case a device broadcasts the a ”GetParents”-message. The device which receives such an wrong list, recovers from multiple device failure in an incorrect way, hence decreasing the connectivity of the key graph due to Theorem 4.1 by one. Hence, the attacker could decrease the key graph connectivity below s after r device failures.

Even though the second attack could be prevented by e.g. a voting mechanism in order to prove the correctness of the send parents-set, the first much simpler attack is almost impossible to

detect. This is due to the fact that it is near impossible to deterministically detect an *Mallory*-type attacker. Hence there is also no way for forcing him to establish keys. Hence, a *Mallory*-type attacker will always be able to reduce the key graph connectivity by one with each device failure.

As shown above a *Mallory*-type attacker has numerous ways to attack our algorithms. While some of these attacks hinder directly the key establishment protocols, others reduce the key graph connectivity by the number of attackers or in the worst case by one with every device failure.

4.3.3 Summary

In this section we analyzed the different possibilities for *Trudy* and *Mallory*. We provided effective countermeasure for our algorithms in order to make our approach resilient against *Trudy*. However, even with these modifications it is still possible for an intruder to hinder the correct execution of our algorithms i.e. perform a denial-of-service attack. Nevertheless, neither active attacker, i.e. *Trudy* and *Mallory*, cannot gain access to secrets, i.e. to newly established keys or encrypted messages – as long as they cannot subvert more than s devices.

4.4 Summary

In this chapter we extended our basic approach towards fault tolerance and active attackers. We introduced the redundancy level of our network – a parameter which can be freely chosen (just like the security level s) according to the specific requirements of a network deployment. With our fault tolerant approach and some given values for s and r we can tolerate up to r device failures at the same time, and up to $s-1$ *Eve*- or *Trudy*-type attackers. We achieved that by modifying our basic approach, hence all the properties for the basic approach still apply.

However, our fault tolerant approach relies – just as the basic approach – on a path search algorithm in order to establish new keys. As already discussed, such a path search might use many resources. Therefore we present in the next chapter an extended approach which overcomes this, and is therefore even more suited for resource-constrained devices.

The fault tolerant approach presented here has been published in [WHCM04] and presented at the conference Pervasive 2004.

5

Extended Key Distribution Scheme

Until now, all our presented approaches for establishing pairwise keys between arbitrary devices rely on a standard path search algorithm for finding s device-disjoint paths in order to establish a new key. However, as we showed in Section 3.5 this might lead to relative high overhead on the devices. Hence, we present in this chapter a novel approach for establishing pairwise shared keys, that does not require such a path search algorithm.

Our new approach is an extension of our basic approach presented in Chapter 3. For the sake of simplicity, in the description of our extended approach we consider only *Eve*-type attackers and no device failures. However, with small adaptations the mechanisms from the fault-tolerant key establishment can be applied also to our new algorithms as discussed in Section 5.7.

The extended approach is based on two observations: Firstly, in our application scenario, the key graph can be modified by temporarily introducing new key edges to it. Secondly, if two nodes in the key graph share s common neighbors, discovering the s node-disjoint paths is trivial, because the source node (i.e. the node initiating the key establishment) only needs to communicate with its direct neighbors.

The extended key distribution scheme consists – just like the basic approach - of two parts: The key graph construction algorithm and the key establishment algorithm. As before, the key graph construction algorithm is responsible for building the initial key graph and for establishing an initial set of key edges whenever a new node is added to the graph. The key establishment algorithm works on graphs built using this algorithm and establishes a key edge between any pair of nodes on demand.

We start out by giving an overview of the extended key establishment algorithm in the next section. After that, details and an analysis of the extended key establishment algorithm are presented in Section 5.2. Based on that analysis, the key graph construction algorithm is then

derived in Section 5.3. In Section 5.4 we prove the correctness of this algorithms. In Section 5.5 we extend our previously introduced methods for the out-of-band key pre-distribution, i.e. the bootstrapping of our network. The different key graph structures, which are possible due to the extended key graph construction algorithm are presented in Section 5.6. Fault-tolerant extensions and the resilience of the extended key distribution scheme against *Trudy*-type attackers are discussed in Section 5.7. Finally we conlcude this chapter with a short summary.

5.1 Overview

The fundamental concept behind the extended key establishment algorithm is to augment the key graph by temporarily inserting additional key edges. More precisely, our goal is to add edges such that there are s paths between source and target node that have exactly one intermediary node, i.e. there exist s 2-hop paths between source and target. After that, the key establishment is straightforward.

The s 2-hop paths are established by forwarding the key establishment query recursively through the network, as follows: The source device sends a ”KeyEstablishmentQuery”-message to all key graph neighboring devices. The devices receiving this query now check if they have a shared key with the target device. If this is the case, the device will respond with an ”Established”-message which is sent to the querying device. If the device does not have a shared key with the target device, it will recursively forward the query to its neighbors. The recursion ends, if the query reaches a device which shares a key with the target device. Due to this recursion we also refer to the extended key establishment algorithm as the *recursive* key establishment algorithm (RKEA). After receiving at least s ”Established”-messages, a device can establish a new key to the target device using the devices which sent the ”Established”-message as the intermediary devices.

Whenever a new key is established by this method, the device that established the new key (possibly as a recursive query) sends an ”Established”-message to all devices it has outstanding ”KeyEstablishmentQuery”-messages from. This way, requesting devices are notified of the new key and the ”Established”-messages propagate back to the source.

In addition to the ”Established”-message, the device that established the key sends a ”CancelQuery”-message to all neighbor devices from which it got an ”Established”-message. This ”CancelQuery”-message informs the devices that they can remove the established key with the target device, since it was used to establish a new key and is not needed anymore. This ”clean-up” is important, since otherwise all devices of the network would almost simultaneously establish keys to the target device – which is not in line with the highly constrained memory on the devices.

The query is finished when the source gets at least s "Established"-messages (and thus can establish a key to the target). Like all other requesting devices, the source device then sends the corresponding "CancelQuery"-messages.

The "CancelQuery"-messages ensure that all temporary keys are removed right after they have been used and the data structures corresponding to this query are removed from the devices.

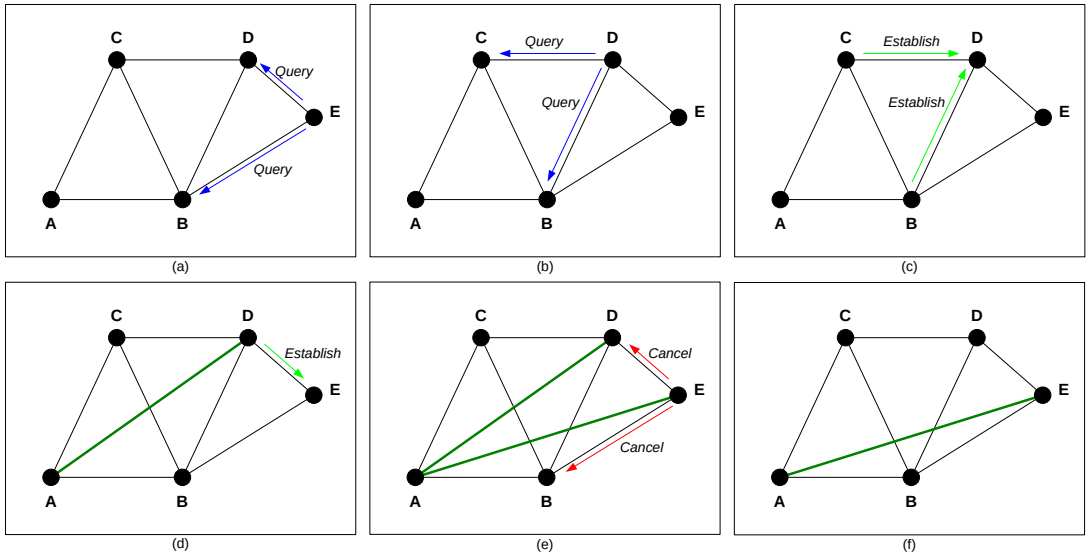


Figure 5.1: Recursive Key Establishment Algorithm Example ($s = 2$)

In Figure 5.1 we present an example for the recursive key establishment algorithm for $s = 2$. In this figure device E (the source device) needs to establish a key to device A (target). Note that green bold lines represent newly established keys. Device E sends a "KeyEstablishmentQuery" to its neighbors B and D (Figure 5.1(a)). Device B has a key to the target device A, and responds with an "Established"-message. Device D, however, does not have a key to A yet, and forwards the query to B and C (Figure 5.1(b)). Both, device B and C have a key to the target device A, and will respond with an "Established"-message sent to D (Figure 5.1(c)).

Device D has now received $s = 2$ "Established"-messages and can itself establish a new key to device A by using B and C as the s distinct intermediary devices.

After establishing this new key, device D sends an "Established"-message to device E and a "Cancel"-message to B and C. Device B and C will ignore this message since they did not actually establish a new key to the target (Figure 5.1(d)).

When device E receives the "Established"-message from device D, it has also received $s = 2$ "Established"-messages and establishes a new key to device A with D and B as intermediary devices. After source device E established a key to the target device A, it sends "Cancel"-messages to B and D. As before, device B will ignore this message. D, however, now removes the key to A since it was needed only temporarily in order to help device E establish a new key.

5.2 Key Establishment

In this section we give a detailed description of our key establishment algorithm. We start by introducing some additionally data structures in Section 5.2.1. Then, in Section 5.2.2, we describe the individual procedures which together form the key establishment algorithm. Finally we conclude in Section 5.2.3 by analyzing the presented key establishment algorithm. This analysis will show some properties of the key establishment algorithm which need to be considered when presenting the algorithm for constructing the key graph in the next section.

5.2.1 Additional Data Structures

In Section 2.5 we already introduced some data structures which are needed for the presentation of our procedures. Before we describe our extended algorithms, namely the key establishment algorithm and the key graph construction algorithm, we need to introduce some additional data structures.

***QueryID*:** The *QueryID* is a network wide unique identifier for a single key establishment query. It is comprised of the triple {*SourceID*, *TargetID*, *Counter*}. *SourceID* is the ID of the device initiating a key establishment and *TargetID* is the ID of the device to which the initiating device needs to establish a key. The last element, *Counter*, is used for unique identification of the current query and set by the source. Each element in a *QueryID* can be accessed using the object oriented dot-notation, e.g. *QueryID.SourceID* gives access to the *SourceID* of this specific *QueryID*.

***EstablishedSet*:** Each device needs to store this set for each query it received (see also *QuerySet*). This set contains a list of device IDs which answered that they share a key with the target (*TargetID*). The following methods are available for this set:

***Add*:** With this method, a new device ID can be added to the *EstablishedSet*. This method takes as a single argument the device ID which needs to be added. In case the ID is already part of this set, nothing happens.

***Remove*:** With this method, a device ID can be removed from the *EstablishedSet*. This method takes as a single argument the device ID which needs to be removed.

***RequestingSet*:** Each device needs to store this set for each query it received (see also *QuerySet*). This set contains initially a list of all device IDs from which a query with a certain *QueryID* was received. The following methods are available:

***Add*:** With this method, a new device ID can be added to the *RequestingSet*. This method takes as a single argument the device ID which needs to be added. In case the ID is already part of this set, nothing happens.

***Remove*:** With this method, a device ID can be removed from the *RequestingSet*. This method takes as a single argument the device ID which needs to be removed.

***QuerySet*:** Each device in the network manages its own instance of this set. This set contains all queries which the device has seen so far and which are still active. A query is a triple of the form { *ID*, *EstablishedSet*, *RequestingSet* }. The element *ID* is the *QueryID* for this specific query. The *EstablishedSet* and the *RequestingSet* are two sets containing device IDs for this specific query as described before. Each entry in this set can be addressed directly by giving the *QueryID*, i.e. *QuerySet*[*someid*].*EstablishedSet* gives access to the *EstablishedSet* of the query with the ID *someid*. The following methods are available for *QuerySet*:

***IDSet*:** This method returns a set containing all device IDs from all entries stored in the *QuerySet*. This method takes no arguments.

***Add*:** With this method it is possible to add a new entry to the *QuerySet*. It takes as a single argument the *QueryID* for which a new element needs to be created. The two corresponding sets are initialized with the empty set. If there exists already an entry with the same *QueryID*, this entry is overwritten and thus all information about the former entry is lost.

***Remove*:** With this method a element from the *QuerySet* can be removed. It takes as a single argument the *QueryID* of the element which needs to be removed.

5.2.2 Algorithm

The recursive key establishment algorithm (RKEA) of our extended approach comprises three procedures: The procedure called when a new ”KeyEstablishmentQuery”-message is received (`onKeyEstablishmentQuery`), the procedure called when an ”Established”-message is received in response to such a message (`onEstablishment`) and the procedure called when a ”CancelQuery”-message is received (`onCancelQuery`). The following sections describe these procedures in detail.

Receiving a ”KeyEstablishmentQuery”-message

The procedure `onKeyEstablishmentQuery` is called whenever a device receives a ”KeyEstablishmentQuery”-message from a key graph neighboring device. Note that a key graph neigh-

boring device is a device with which this device shares a key (there is an edge in the key graph between the corresponding nodes).

```
1:  procedure onKeyEstablishmentQuery(Sender, QueryID)
2:  if QueryID.TargetID ∈ (KeyStore.IDSet ∪ EstablishmentKeyStore.IDSet) then
3:     if Sender = MyDeviceID then
4:        // key already established, no need to engage the algorithm;
5:        return
6:     else
7:        SecureSend(Sender, Established, { QueryID });
8:     end if
9:  else
10:    if QueryID ∈ QuerySet.IDSet then
11:       // we encountered this search already, so we just remember which device queried
12:       QuerySet[QueryID].RequestingSet.Add(Sender);
13:    else
14:       // this is a new query
15:       QuerySet.Add(QueryID);
16:       QuerySet[QueryID].RequestingSet.Add(Sender);
17:       for all DeviceID ∈ KeyStore.IDSet do
18:          SecureSend(DeviceID, KeyEstablishmentQuery, { QueryID } );
19:       end for
20:    end if
21: end if
```

Procedure 5.1: Receiving a ”KeyEstablishmentQuery”-message

The pseudo-code for this procedure is shown in Procedure 5.1. When the procedure is called it first checks if it already shares a key with the target device by looking it up in the main *KeyStore* (line 2). If a key is found and this query was not initiated locally (line 3), an ”Established”-message is sent back to the requesting device (line 7).

If no key is found, the device checks if it encountered this query before (line 10). If this is the case, *Sender* is added to the *RequestingSet* for this specific *QueryID*. This makes it possible to notify all requesting neighbors if and when a key from this node to the target was established successfully.

If the device has not encountered this query yet, it adds a new entry to the local query database (*QuerySet*). When adding the new query, the corresponding *EstablishedSet* is initialized with the empty set and the *RequestingSet* with the *Sender* from which this query was just received. Since this device does not share a key to the target yet, it forwards this query to all of its neighbors (line 17ff).

When a device needs to establish a new key to another device, it initiates a new query by calling the procedure `onKeyEstablishmentQuery` locally providing its own ID as the *Sender*.

Receiving an "Established"-message

A device that shares a key with the target device of a query responds to a "KeyEstablishment-Query"-message with an "Established"-message. A device that receives such a message can therefore deduce that its *Sender* shares a key with the target of that query.

```
1:  procedure onEstablishment(Sender, QueryID)
2:  if QueryID ∈ QuerySet.IDSet then
3:     QuerySet[QueryID].EstablishedSet.Add(Sender);
4:     QuerySet[QueryID].RequestingSet.Remove(Sender);
5:     if |QuerySet[QueryID].EstablishedSet| ≥ s then
6:        if (|QuerySet[QueryID].RequestingSet| > 0) ∨
          (QueryID.SourceID = MyDeviceID) then
7:           if SimpleKeyExchange(QueryID.TargetID) then
8:              for all DeviceID ∈ (QuerySet[QueryID].RequestingSet - {MyDeviceID}) do
9:                 SecureSend(DeviceID, Established, { QueryID });
10:             end for
11:             for all DeviceID ∈ (QuerySet[QueryID].EstablishedSet - {MyDeviceID}) do
12:                SecureSend(DeviceID, CancelQuery, { QueryID });
13:             end for
14:          end if
15:       end if
16:    end if
17: end if
```

Procedure 5.2: Receiving a "Establishment"-message

The pseudo-code for the corresponding procedure `onEstablishment` is shown in Procedure 5.2. The device first checks if the "Established"-message it received belongs to an active query (line 2) – if not, the message is ignored. The sender of the message is added to the *Established-Set* for this query and removed from the *RequestingSet* if necessary (line 3ff).

The *EstablishedSet* holds all device IDs which share a key with the target. If the *EstablishedSet* contains at least *s* entries, the corresponding devices can be used as intermediaries to establish a new shared key (line 5). However, establishing a new key to the target device is only necessary if the *RequestingSet* is non-empty or the device is the source device of the query (line 6).

If the new key establishment is successful (line 7), the device has to announce this to all devices who requested the key establishment, namely the devices stored in the *RequestingSet*.

Finally, a ”CancelQuery”-message is sent to any device in the *EstablishedSet*. This is needed for the ”clean-up” described in the next subsection.

Receiving a ”CancelQuery”-message

The ”CancelQuery”-message is used to remove all temporary keys and local data for a certain query throughout the network. Receiving a ”CancelQuery”-message for a query from a certain device implies that this device has no longer an interest in the establishment of a key to the target of this query.

```
1:  procedure onCancelQuery(Sender, QueryID)
2:  if QueryID ∈ QuerySet.IDSet then
3:     if Sender ∈ QuerySet[QueryID].RequestingSet then
4:        QuerySet[QueryID].RequestingSet.Remove(Sender);
5:        if QuerySet[QueryID].RequestingSet = ∅ then
6:           removeKey(QueryID.TargetID); // agree with target device to remove this key
7:           QuerySet.Remove(QueryID);
8:        end if
9:     end if
10: end if
```

Procedure 5.3: Receiving a ”CancelQuery”-message

The pseudo-code for the corresponding procedure is given in Procedure 5.3: When receiving a ”CancelQuery”-message, the *Sender* of the message is removed from the *RequestingSet* (line 4). In case the *RequestingSet* has become empty through this action, all data about the corresponding query is removed from the local database (line 7). Additionally, a key which has been established in the context of this query is removed (line 6).

5.2.3 Analysis

With the algorithm presented in this section, we can establish a new shared key between two arbitrary devices. This can be done without explicitly searching for s device-disjoint secure paths. Unfortunately, the above presented algorithm does not work on all s-connected key graphs.

To see the situation that might occur, consider the following case presented in Figure 5.2. The key graph for this network was constructed using the algorithm from Section 3.2.1. However, if we apply the above key establishment algorithm to this key graph, the algorithm will end in a deadlock situation: Device A needs to establish a new key to device J. In order to do so, it sends a ”KeyEstablishmentQuery”-message to E, C, and B (Figure 5.2(a)). None of the devices

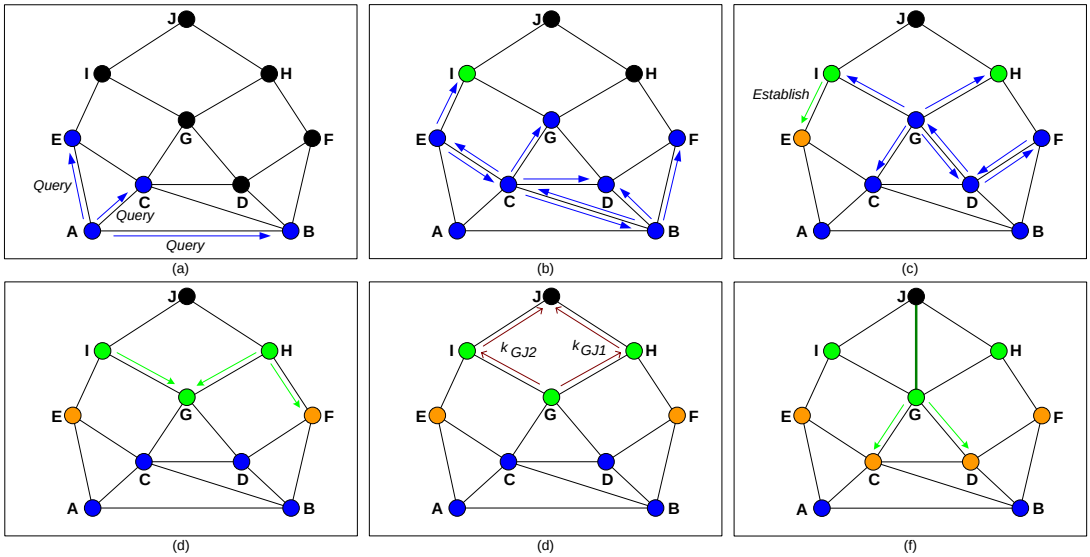


Figure 5.2: Key establishment deadlock

so far have a key with the target J, so they all forward the ”KeyEstablishmentQuery” to their respective neighbors (Figure 5.2(b)).

The first device which has a key with the target device and is reached by the query is I. I will respond to E with an ”Established”-message. In accordance with Procedure 5.2, E adds I to its *EstablishedSet* (Figure 5.2(c)). Analogously, device F adds H to the *EstablishedSet*. The *EstablishedSets* of device E and F now contain exactly one element, and both devices wait for more ”Established”-messages.

Device G gets two ”Established”-messages (from I and H), and can therefore establish a new key to J (Figure 5.2(d)). G now sends two key shares via I and H to J, thus establishing a new key with J (Figure 5.2(e)).

Device G announces the newly established key by sending an ”Established”-message to C and D (Figure 5.2(f)), which will add G to their *EstablishedSet*. At this point, the algorithm has reached a deadlock: Devices E, C, D and F have exactly one entry in their *EstablishedSets*, and are waiting for additional ”Established”-messages. However, no device can establish a new key to the target and no further ”Established”-messages will be sent.

A deadlock occurs if no device in the graph can establish a new key to the target device. The recursive key establishment algorithm will *never* deadlock[1] if there is always a set of s devices that share a key with the target device and for this set, a device exists that shares a key with all devices in the set, but not the target device. If this can be assured for any pair of source and destination devices, the key establishment algorithm will always be able to establish a

[1] For a full proof see Section 5.4.

shared key between them. To ensure this, we need to modify our basic key graph construction algorithm. In the next section, we describe a extended key graph construction algorithm that ensures successful key establishment. Furthermore, we prove in Section 5.4 that the adapted graph structure guarantees this using the RKEA algorithm.

5.3 Key Graph Construction

In this section, we provide an algorithm for constructing a RKEA-compatible key graph, i.e. a key graph which allow the establishment of new keys using the key establishment algorithm as described in the last section. As with the basic approach, our key graph construction algorithm is comprised of two parts: one algorithm for adding a new device to the network and one for removing a device from the network. This means we need an algorithm for adding a node to a (possibly empty) key graph and one for removing a node from a key graph while preserving the necessary properties of the graph.

We describe the algorithm for adding a new device to the network in Section 5.3.1 and the algorithm for removing a device from the network in Section 5.3.2

5.3.1 Adding a New Device to the Network

With the key graph construction algorithm of our basic approach (Section 3.2.1) we showed how to obtain an s-connected graph by construction: Beginning with a fully connected graph for the first s nodes, each further node is connected to s randomly selected nodes of the existing graph. Due to [Men27], this method always yields graphs in which there always exist s node-disjoint paths between any pair of nodes.

This key graph construction method needs to be modified for RKEA. Before describing these modifications we introduce two new definitions.

Definition 5.1 (k-clique) *In a graph $G = (V,E)$ with $n = |V|$ a k-clique with $k < n$ is a subgraph $G' \in G$ of k nodes which is fully connected.*

Definition 5.2 (s-connector) *An s-connector is set of s nodes in a graph, that can be used for a new node to connect to the graph.*

In the basic approach each subset of s nodes is an s-connector. In the extended approach presented here each s-connector is an s-clique. This ensures the successful key establishment, as proven in Section 5.4. Note that a devices' own s-connector corresponds to its parents-set.

The idea of the extended key graph construction algorithm is as follows: Again beginning with a fully connected key graph for the first s nodes, each further node is connected to a clique of s

nodes. Connecting the new node with all s nodes of a clique results in a $(s+1)$-clique. Thus, a graph with n nodes is assembled of $(n-s)$ cliques each of $(s+1)$ nodes.

Algorithm 5.1 Extended key graph construction – adding a node

1: Given a graph $G_K = (V, E_K)$ with $n = |V|$ with nodes $v_i \in V$ and a new node v_{n+1}
2: $V := V \cup \{v_{n+1}\}$;
3: **if** $s \geq n$ **then**
4: **for** $i = 1$ to n **do**
5: $E_K = E_K \cup \{v_{n+1}, v_i\}$; // provide a new pairwise key securely
6: **end for**
7: **else**
8: $C := \{c_1 \dots c_s \in (V - \{v_{n+1}\}) | \forall c_i, c_j$ with $i, j \in [1 \dots s] \wedge i \neq j : \exists (c_i, c_j) \in E_K\}$
9: **for** $i = 1$ to s **do**
10: $E_K = E_K \cup \{v_{n+1}, c_i\}$ with $c_i \in C$; // provide a new pairwise key securely
11: **end for**
12: **end if**

The extended key graph construction algorithm is given in Algorithm 5.1. As already mentioned, the case where there are fewer than s nodes in the key graph is the same as in the basic approach, hence there is no change in the first part of the algorithm. However, when there are more than s nodes in graph and we add a new node, we must choose an s-connector in order to introduce this new node to the key graph. Thus, the only difference in this algorithm with respect to the basic adding algorithm (Algorithm 3.1) is in line 8. Here we choose an valid s-connector C, i.e. we choose s nodes $c_1 \dots c_s$ in such a way that for all nodes c_i, c_j with $i, j \in [1 \dots s]$ there exists an edge $(c_i, c_j) \in E_K$. Informally this means that we choose a fully connected subset of s nodes from all nodes, hence an s-clique.

Note that an *s-connector* can always be found when introducing a new node to a graph with $n > s$ for the following reasons:

- The first s nodes will be fully connected, thus forming a clique – the first s-connector.
- By having an edge to every node of an s-connector, each newly introduced node itself introduces s new s-connectors.

Hence with $x \geq s$ nodes we get $1 + (x-s)s$ available s-connectors, thus an s-connector can always be found when introducing a new node.

Figure 5.3 shows the resulting graph structures for $s = 2$, $s = 3$, and an abstracted view for an arbitrary s. As can be seen in Figure 5.3(a), for $s = 2$ the resulting graph is a planar graph composed only of triangles. The triangles are $(s+1)$-cliques, while each side of any triangle

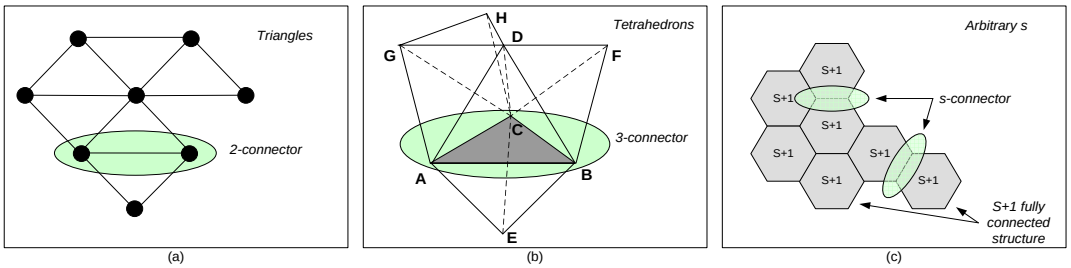


Figure 5.3: Graph structures when using only s-connectors for introducing new nodes

represents a 2-connector (a 2-clique). For $s = 3$, it is easiest to imagine a three-dimensional structure (Figure 5.3(b)). The $(s+1)$-cliques, i.e. the 4-cliques can be represented by tetrahedrons in 3D, while the 3-connectors are the 4 triangles of each tetrahedron. Generalizing this concept to an arbitrary s, we always get a graph composed of $(s+1)$-cliques, and each of these cliques has s s-connectors (s-cliques) (Figure 5.3(c)).

In Section 5.4, we show that RKEA can always establish a shared key for a key graph constructed in this fashion. What remains to be discussed is how such a graph structure can be sustained when removing a node from the graph. The next subsection discusses this issue.

5.3.2 Removing a Device from the Network

Removing a device from the network can destroy the s-connected property of the corresponding key graph. An algorithm that removes a device in a controlled shutdown procedure, i.e. one in which a device that disconnects from the key graph makes all necessary arrangements before leaving, is described in the following. The case of a device failing and thus having no time to make all the arrangements is discussed in Section 5.7. Observe that for RKEA, both the s-connected property of the key graph and the resulting structure of $(s+1)$-cliques has to be preserved. Therefore, we divide the removal procedure in two steps: Preserving the s-connected property and preserving the $(s+1)$-cliques.

Preserving an s-connected key graph

In Section 3.2.2, we presented an algorithm for removing a device from the network while preserving the s-connected property of the underlying key graph. We showed that any node which misses one of its original edges needs to add new edges to a node which was introduced before it. We further concluded that all nodes in the parents-set of a node have been introduced before the node itself, hence they are good candidates for replacing the missing edge. Hence, our extended solution will include the concept for removal of a node from the basic approach.

Preserving the $(s+1)$-cliques of the key graph

Whenever a device leaves the network, s s-connectors are going to be destroyed in the key graph. Each device which used such an s-connector needs to re-establish keys in order to connect again to a correct s-connector.

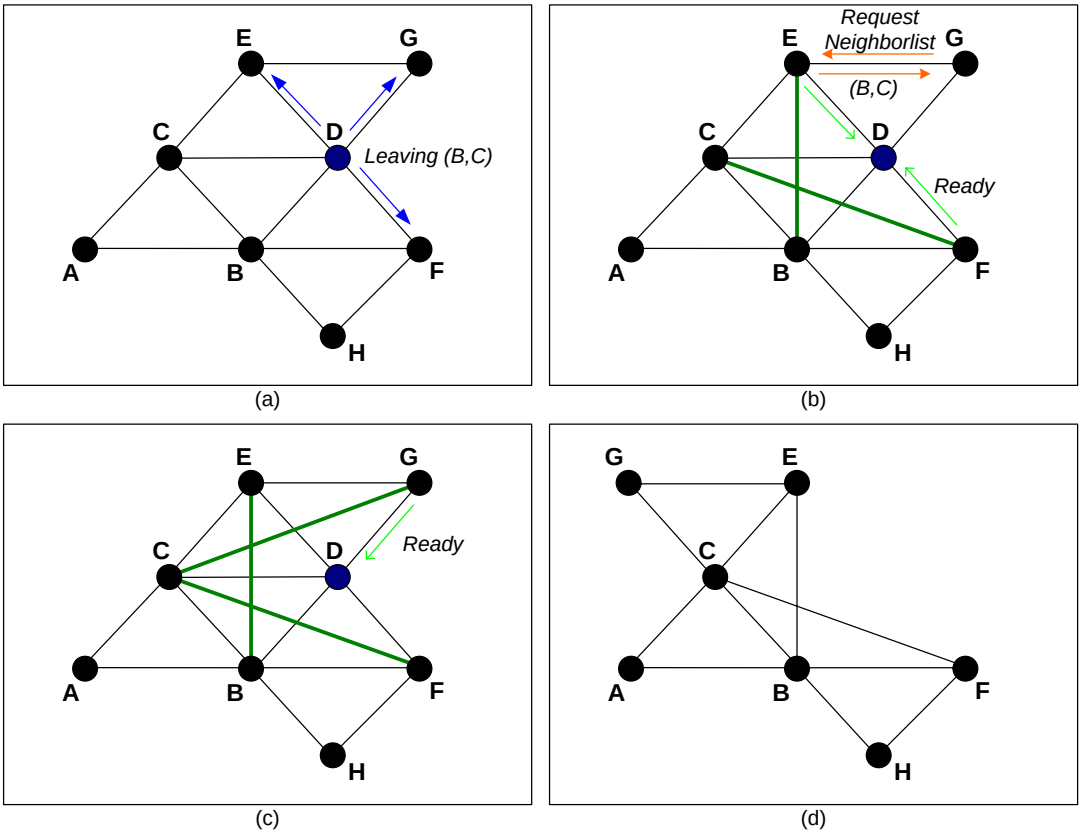


Figure 5.4: Removing a device from the network ($s = 2$)

Consider the example in Figure 5.4. Device D wants to leave the network and sends a "Leaving-Intention"-message to all devices from its children-set (Figure 5.4(a)). The "LeavingIntention"-message includes the devices of the s-connector which D used to enter the network. The devices receiving this message compare the received s-connector to their own parents-set, i.e. their s-connector.

Those devices which find an $(s-1)$-match between their own s-connector and the received s-connector can easily identify the device to which they have to establish a new key: It is the only device which is contained in the received s-connector but not in their own.

In Figure 5.4(b) devices E and F are in this situation. By comparing the s-connectors, i.e. $\mathcal{P}_{v_D} = B,C$ and $\mathcal{P}_{v_D} = C,D$, E finds that it needs to establish a new key to B. Analogously F finds that it needs to establish a key with C.

Meanwhile device G has no match at all when comparing the received parents-set with its own parents-set. In such a case G needs to ask a device from its own s-connector (i.e. parents-set) for that device's s-connector – this is the same procedure as when a new device is introduced. Note that this mechanism ensures also the s-connectivity of the key graph since we did choose an older node for replacing the missing edge. In the example, there is only node E to ask, which responds with the list (B,C) (after E is done with re-establishing keys itself). From this list, G can choose a partner randomly (in the example, C).

After establishing new keys, all affected devices send a *Ready*-message to D (Figure 5.4(c)). When D has received a *Ready*-message from all devices in its $\mathcal{C}$-set, it can safely leave the network – all properties have been restored (Figure 5.4(d)).

In general we can distinguish two cases. The first case is actually a special case and therefore only an optimization. In this case the received parents-set of the leaving device matches in $s-1$ IDs. Hence, in order to recover, a device picks the only ID which is not yet in its own parents-set.

The second case is everything else. The simplest way for a device A to recover in this situation is to request the parents-set of a remaining device from the own parents-set, i.e. $X \in \mathcal{P}_A$. The received parents-set $\mathcal{P}_X$ together with the device X form a $s+1$-clique. Hence, there are $(s+1)$ s-connectors available for recovery. Since one device is already in our parents-set (X), we need to establish $s-1$ new keys with nodes from $\mathcal{P}_X$, in order to be connected to a different s-connector. The old keys to the old s-connector can then be removed.

5.4 Proof of Correctness

In this section we prove by induction over n (total number of devices) that in a network constructed with the extended key graph construction algorithm (Algorithm 5.1) RKEA can always establish a new key between any pair of devices.

Induction Statement In a graph constructed with the extended key graph construction algorithm, every node can construct a new edge to any other node using the recursive key establishment algorithm.

Induction Start, $n = s+2$ We start with $n = s+2$ since $s+1$ nodes are always fully connected, and therefore no new edge is needed. Moreover, when introducing a new node to a graph consisting of $(s+1)$ nodes, it does not matter which s nodes are used, since every set of s nodes is fully connected and therefore forms a correct s-connector (see also Figure 5.5).

In a graph with $n = s+2$ nodes there are only two nodes which do not share an edge. These two nodes share s common neighbors. Hence, when the corresponding device needs to establish a new key, it sends an "KeyEstablishmentQuery" according to Procedure 5.1. Since all s neighbors have a key to the target device, all devices will respond with an "Established"-message, thus enabling the device to establish a new key.

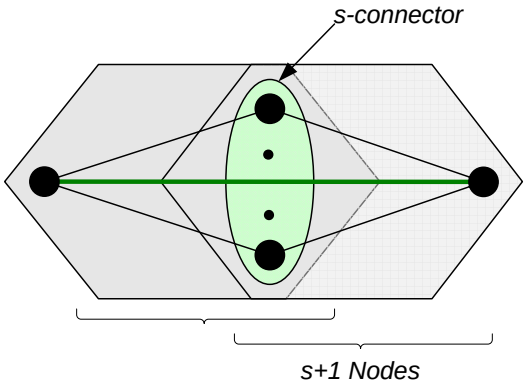


Figure 5.5: Induction Start: $n = s+2$

Induction Step Suppose the induction statement is proved for all graphs constructed using Algorithm 5.1 with at most n nodes. Consider an arbitrary graph of n nodes in which the induction statement is true. We now introduce a new node Y by connecting it to an s-connector (Figure 5.6).

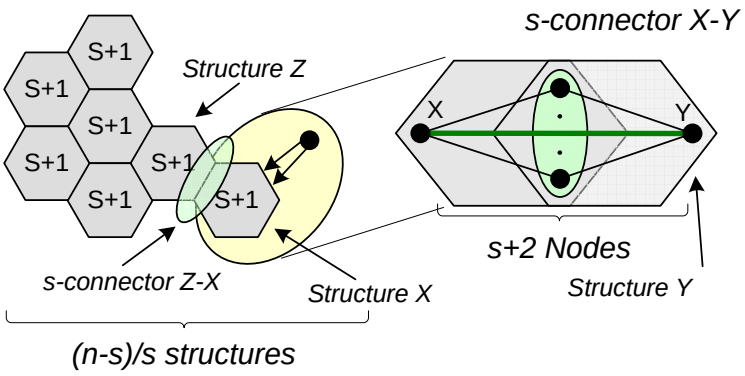


Figure 5.6: Induction Step: Adding one node

Observe that since queries are forwarded to each neighbor, even for the same source-destination pair, queries forwarded from different nodes do not influence each other. Hence, the sequence in which queries are forwarded and answered is not relevant to the algorithm. This also means that there only needs to be one successful sequence of key establishment queries for a successful key establishment using RKEA. Especially, for any key graph G, if RKEA can establish a key

between two nodes in a subgraph $H \subseteq G$, then RKEA can also establish a key between these nodes in G.

Suppose that for the new graph, an arbitrary node A attempts to establish a key to any node B using RKEA. Three cases have to be examined:

Case 1: $(A \neq Y) \wedge (B \neq Y)$. In this case, by the induction statement, the subgraph *without* Y is sufficient to establish a key between A and B.

Case 2: $A = Y$. Using RKEA, all nodes in the s-clique that Y is connected to receive a "KeyEstablishmentQuery" from Y. Since the query is forwarded recursively, by the induction statement, all these s nodes will establish a key to B, allowing Y to establish a key with B via these nodes.

Case 3: $B = Y$. Recall from Section 5.3.1 that the key graph consists of $(s+1)$ cliques connected by s-cliques. Without loss of generality, consider the subgraph obtained by only examining the shortest sequence of $(s+1)$-cliques from the one containing A to the one containing Y: By construction, the s nodes to which Y did initially connect are part of an $(s+1)$-connector. Thus, there is a node X that is part of this $(s+1)$-connector, but does not share a key with Y (Figure 5.6). X will eventually receive a "KeyEstablishmentQuery", and establish a key with Y via the s other nodes. By doing so, an $(s+2)$-connector is formed.

Now, remove a node from this $(s+2)$-connector that is neither X nor Y nor part of the s-connector joining X's s-connector to neighboring node Z's s-connector (Figure 5.7). Note that the graph obtained in this fashion is still a graph that could have been generated by Algorithm 5.1. However, this graph only has $m \leq n$ nodes, and all nodes have received and forwarded a "KeyEstablishmentQuery" for nodes A and Y. Hence, by the induction statement, a key between A and Y can be established.

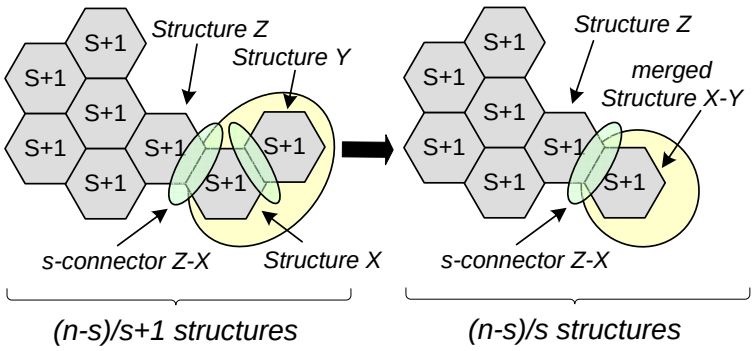


Figure 5.7: Induction Step, case 3: Merging cliques

5.5 Extended Bootstrapping: Creating the Initial Key-Graph

In the last sections, we presented an extended algorithm for obtaining our key graph and establishing new keys between devices. In addition, we discussed in Section 3.4 different approaches for creating the initial key graph. Remember, that the initial keys need to be distributed over a secure channel, i.e. out-of-band. However, due to the extended requirements of this approach we need to adapt our basic bootstrapping approaches. In the following we describe the extensions to the already introduced bootstrapping possibilities.

5.5.1 Static Pre-Distribution

The classical pre-distribution works similar as described before with the basic algorithm. Hence, the manufacturer of a network stores the initial key sets on the devices before deploying the network. The only difference to the basic bootstrapping method is, that the manufacturer now needs to store the keys on devices using Algorithm 5.1 and therefore creating a key graph which is compatible with our recursive key establishment algorithm. Thus, this approach has the same advantages and disadvantages as the basic bootstrapping approach which is described in Section 3.4.1.

5.5.2 Configuration Device

As before the usage of a configuration device is another possibility to introduce new devices to a network. We presented already in Section 3.4.2 the basic principle of such a configuration device.

In the basic bootstrapping the configuration device only needed to pick randomly s devices to place a key with a new device. However, in the extended bootstrapping process it cannot choose completely randomly since it needs a valid s-connector.

Hence, the question remains how such an s-connector can be found by the configuration device. At least two possibilities exist. The simplest method is that the configuration device stores a global view of the network. This can be done by always remembering the device IDs which have been used when introducing a new device. However, the global view would become inconsistent if devices are removed without also updating the global view of the configuration device. Due to this drawback we favor the following approach: Whenever a new device is introduced into the network, the configuration device picks one device from the network randomly. As already mentioned, each device is part of an $(s+1)$-clique, hence it has knowledge about s different s-connectors with the device itself as a part of the specific s-connector. The configuration device requests one of theses s-connectors from the device. The device might

randomly choose one s-connector, i.e. the remaining $(s-1)$ device IDs, and then send this information back to the configuration device. With this information, the configuration device can now complete the introduction of the new device.

The configuration device method for bootstrapping has the same advantages and disadvantages as described in Section 3.4.2. With the extended algorithm, there is an additional communication overhead for communicating a valid s-connector.

5.5.3 Physical Contact

In Section 3.4.3 we introduced also a user-driven method, where the user is responsible for exchanging the first s keys for a new device. The preferred method for exchanging a new key between two devices by the user was to bring the two devices in physical vicinity and then initiate the key exchange over a secure channel (e.g. a cable). However, this method is based on the fact that the user could choose s devices randomly, which is no longer true for our extended approach. With our extended approach, the user must choose a valid s-connector. Since the user cannot be expected to know which set of devices form together an s-connector, we need some additional mechanisms to help the user.

In the following we describe two possibilities for solving the extended bootstrapping problem with physical contact. First, we describe a simple method which is very similar to our extension in the last section with the configuration device. Second, we describe a more complex method, where the user again is free to use s random devices.

Manual

Whenever a device is introduced in the network, the user randomly chooses a device already in the network and establishes a new key in an out-of-band secure way e.g. via a secure physical connection. From this point on we proceed similar as in the method with the configuration device: The contacted device from the network sends the IDs of $s-1$ devices that form together with the device itself an s-connector to the new device. The new device displays them to the user, e.g. on a small display, who can now choose further $(s-1)$ devices and finish the introduction process in the same way as for the first device. Having established a new key to the other devices, the new device is fully introduced in the network and can use our recursive key establishment algorithm to establish any further keys.

Consider the example from Figure 5.8. Device H needs to be introduced to the already existing network (devices $A-G$). Therefore the user first exchanges a key with the randomly chosen device F by physical contact. After having this first key, H can communicate securely with F and requesting the first s devices from F (Figure 5.8(a)). F responds with B and D (Figure 5.8(b)).

This information needs to be communicated to the user, e.g. by displaying it on a display. Now, the user knows with which other device he needs to further establish a key in order to introduce the new device. In the example the user chooses device *B* (Figure 5.8(c)).

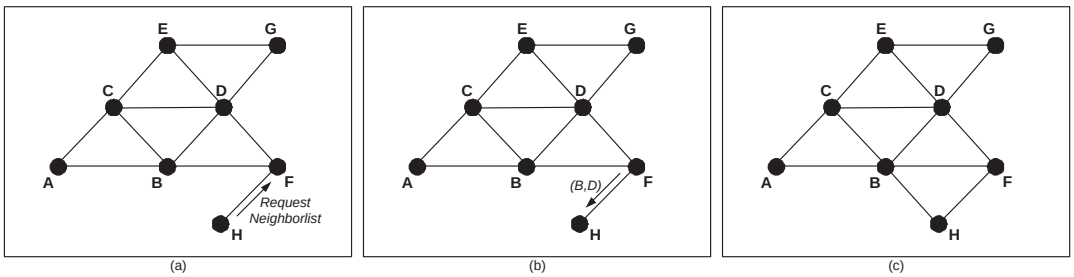


Figure 5.8: Introducing a new device to the network ($s = 2$)

Automatic

After choosing randomly the first device with the manual physical contact method, the user needs to use $(s-1)$ specific other devices for introducing a new device. However, some of those devices might not be easy reachable. Therefore we developed a small extension, which allows the user again to choose all s devices randomly.

The automatic physical contact method is based on the fact that our RKEA itself can be used to correct the primary key set on a newly introduced device. With this method the user establishes new keys with s randomly chosen devices of the network. Even if this results in the new device *not* being connected to a valid s-connector, the new device can request as before a valid s-connector of one of the devices to which it already has a key. After obtaining this information, it can – just like in the manual case – establish new keys to this s-connector, but unlike in the manual case, using RKEA. Having established new keys with a valid s-connector, the initial keys exchanged by the user can be removed. Hence, the difference to the manual case is that the user only needs to exchange s keys when introducing the new device and the device itself will establish keys to a valid s-connector.

The automatic approach – using RKEA with a device which is not connected to a valid s-connector – is sound for the following reason: The initial contacted devices are already part of the network and can therefore establish a key with any other device of the network. When these devices have all established a key to the same device, the new device can do so as well, since it now has s intermediary nodes to the device. Thus a device, even if not yet fully introduced, can establish new keys to devices in the existing network – thus establishing keys to an valid s-connector.

5.6 Key Graph Structures

In this section we illustrate different key graph structures, which result from modifying the way how the ”random” s-connector for introducing a new node according to Algorithm 5.1 is chosen. This analysis is important since it influences some properties of our approach. One important property of our approach is the memory consumption on each device, i.e. how many keys need to be stored on a single device. The number of keys on a device correlates directly with the degree of the corresponding node in the key graph.

We distinguish three different key graph structures: *random*, *k-degree*, and *chain*. In the following we discuss each of them briefly.

5.6.1 Random Graph

When introducing a new node into the key graph by using a purely random s-connector we get a *random* key graph. In Figure 5.9 we show an example for a random key graph.

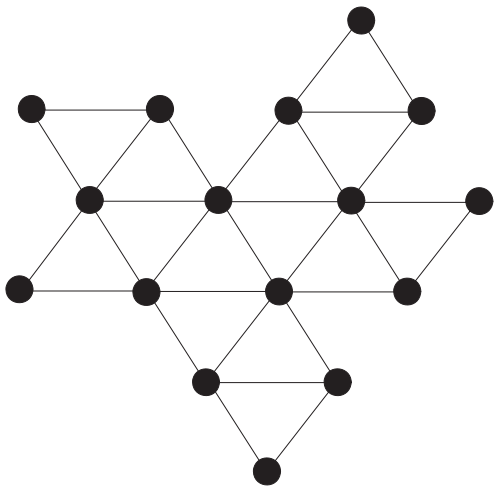


Figure 5.9: Random key graph structure for $s = 2$

Obviously the random key graph offers the greatest freedom when introducing a new node into the key graph. However, it offers no control at all over the degree of a node. In worst case all nodes which are added to the key graph after the s-th node use the first s-connector. Hence the degree of the first s nodes would be $(n-1)$. Hence, the degree is only limited by the total number of nodes in the network and thus the memory consumption could become a bottleneck.

5.6.2 k-degree Graph

In order overcome the drawback of the possible high node degree of the purely random key graph structure we introduce the *k-degree* key graph. We define a k-degree key graph as a key graph where we limit the node degree to a certain maximum $k = d_{max} \geq 2s$. Whenever a new node is introduced into the key graph only such s-connectors are used where $s-1$ nodes have a degree of at most $d_{max}-2$. The remaining node from the s-connector must have a degree of at most $d_{max}-1$. These conditions are needed since otherwise the key graph could reach a state where no s-connector can be used anymore – with these conditions each new introduced node introduces at least one new s-connector. Hence, there will always be a valid s-connector in a network. Figure 5.10 shows such a key graph where a d_{max} of $2s+1=5$ was used. In this figure the dark nodes symbolize devices which cannot accept any further children, hence they cannot be used as part of an s-connector. Only the lighter nodes in the figure, i.e. the nodes with a degree smaller than five can be used by new devices as part of a valid s-connector.

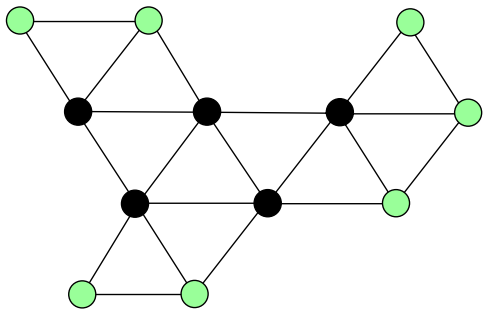


Figure 5.10: k-degree key graph structure for $s = 2$ and $k = d_{max} = 2s+1 = 5$

With the k-degree key graph we can achieve a node degree which is independent from the total amount of nodes in the network, i.e. it is constant. Since the memory on any real device is never unlimited the k-degree approach provides a solution for steering the consumption on each device.

Obviously the method used for creating the initial key graph must be adapted to follow the additional rules for creating a k-degree key graph. With the classical pre-distribution by the manufacturer, the manufacturer only needs to follow the additional rule when distributing the key sets. The configuration device can query different nodes for their degrees in order to find a suitable s-connector. It is little more complicated with the physical contact bootstrapping approach: Here, we need to send a query to the network to which suitable devices respond. This way the user (or in the automated physical contact method the introduced device) gets the knowledge about a valid s-connector.

5.6.3 Chain Graph

The *chain* key graph structure is a special case of the above k-degree key graph structure: a chain key graph structure is a k-degree key graph where $k = 2s$. While this key graph structure represents the best case in terms of node degree, it represents the worst case for communication overhead, since a new key between one end of the chain and the other end uses every other node in the key graph for temporarily keys. In Figure 5.11 we show an example of a chain key graph with $s = 2$.

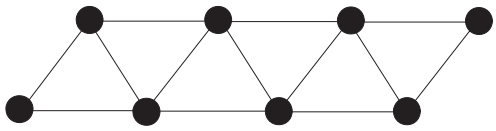


Figure 5.11: Chain key graph structure for $s = 2$

Obviously a chain key graph can be created by the same methods which are used to create a k-degree key graph – we only need to set $k = 2s$. However, due to the special case there exists an additionally possibility to create a chain key graph structure: we always just need to remember the last s introduced devices. Hence, whenever a new node is added to the key graph we add an edge to the last s added nodes. This simple algorithm could even be used by the user and always results in a chain key graph, even though it uses global knowledge.

5.7 Fault Tolerance

In the last sections, we introduced our extended key distribution scheme. In Chapter 4 we introduced our fault tolerant approach, which is tailored towards our basic key distribution scheme. However, the concepts introduced in Chapter 4 can be easily adapted for our extended key distribution scheme.

In the following, we describe these adaptations. We discuss in Section 5.7.1 the necessary adaptations in order to tolerate device failures. The adapted recovery mechanisms are then discussed in Section 5.7.2 and finally in Section 5.7.3 mechanisms for coping with a *Trudy*-type attacker are sketched.

5.7.1 Device Failures

In order to tolerate device failures the same mechanisms can be used as presented in Section 4.2. Hence, we only need to establish a $(s+r)$-connected key graph.

The modified algorithm for the key graph construction can be derived directly from Algorithm 5.1 by replacing each s with $(s+r)$. By executing this algorithm when adding a new node to the graph we get an RKEA compatible key graph which is at least $(s+r)$-connected.

Having an $(s+r)$-connected key graph constructed with the extended key graph construction algorithm our presented recursive key establishment algorithm from Section 5.2 can establish a new key even in the presence of up to r device failures. Due to the way how RKEA works - similar to flooding – there are no modifications needed in order to tolerate device failures. This is due to the fact that there are still at least s devices in the vicinity of each device left which will forward the query.

Hence in order to tolerate r device failures we need to establish a $(s+r)$-connected key graph using Algorithm 5.1.

5.7.2 Key Graph Connectivity Recovery

To recover from a device failure and sustain the connectivity of the key graph, the recovery approach presented in Section 4.2.2 must be adapted to work with the extended key distribution scheme.

In the original recovery approach, if a device fails, all lost edges in the key graph are replaced with edges to nodes in the parent-set of the failed device. In the extended approach this is no longer sufficient, as we cannot ensure that this will result in valid s-connectors. Therefore, each device A that has to replace an edge, contacts a device B from its own parents-set and requests B's parents-set. It then uses this parents-set as an alternative s-connector and replaces its existing parents-set with B's. Note that while this guarantees a valid s-connector for A, it may destroy the s-connector for A's children. Therefore, these devices may have to select a new s-connector, too, using the same approach. This process ends if there are no more children with an invalid s-connector.

This extended approach can recover from up to r simultaneous device failures.

5.7.3 Trudy

Both attacks as described in Section 4.3.1 are not possible with the extended approach. This is simply due to the fact that each step in the recursive key establishment involves s different devices, i.e. each step is s resilient. This means, as long as *Trudy* has not subverted at least s devices there will always be at least one device left which will act correctly and therefore hinder *Trudy* to deceive the algorithm.

If *Trudy* tries to mount a man-in-the-middle attack, she would need to establish a new key to a device A and claiming to be B. While she could fake such a query, she would fail in the last step

where the actual key needs to be established. Since – in contrast to our basic and fault tolerant approach – we use here only paths with a singe intermediary node. Therefore, it is easy for these intermediary nodes to authenticate both sides.

With the above observation it is also clear that *Trudy* cannot mount the Sybil attack. She has no way of establishing a new key from a non-existing device – in order to do that she would need to collaborate with at least s other devices. Hence, as long as she cannot subvert more than s devices, also this attack is not possible.

Any attacks on the recovery algorithm are useless as well. The recovery algorithm itself does not establish any keys, but relies for key establishment on the respective algorithm. Hence, an attack on the recovery algorithm in order to gain access to secrets will lead to an attack on the key establishment algorithm, which is resilient against *Trudy*.

Therefore, as long as *Trudy* can subvert only less than s devices, she cannot intrude the network. She could send any of the three messages used in the key establishment algorithm – it will not gain her a better position in the case of a key establishment between two arbitrary unsubverted devices.

5.8 Summary

In this chapter, we presented our extended approach for key distribution tailored specifically for sensor and actuator networks. It uses only symmetric cryptography and unlike the first two presented approaches it does not rely on a standard path search algorithm for key establishment. Our algorithm guarantees the ability for an arbitrary pair of devices to exchange a key in a secure fashion, provided that the number of devices an attacker – *Eve* or *Trudy* – is able to subvert is not higher than the security level s of the network. This was achieved without prior knowledge of the maximum network size. The network can grow incrementally and also shrink if devices are removed in a controlled fashion.

Our extended approach is very memory efficient - it can be parameterized in a way that is uses only constant memory on all devices, independent of the network size. A detailed evaluation of the extended approach's memory consumptions and communication overhead, as well as a comparison of the performance of the different approaches is given in the next chapter.

The extended key distribution scheme has been published in [WKHR05] and presented at the ACM conference SenSys 2005.

6

Evaluation

This chapter complements the presentation of our key distribution schemes with a detailed evaluation. We start in Section 6.1 by presenting the performance metrics used for evaluating our approach. In Section 6.2 we present the parameters which influence the performance metrics. Equipped with the metrics and the parameters we analytically discuss our approach in Section 6.3. In Section 6.4 we present emulation results to further strengthen our theoretical results. Finally we close this chapter with a short discussion in Section 6.5.

6.1 Performance Metrics

Our main performance metrics are attacker resilience, message overhead, and memory consumption. In the following we provide some details for each of these performance metrics.

Attacker Resilience *res*

With this evaluation we introduce the attacker resilience *res* which is a numerical percentual value for the graceful degradation of the network. We define the attacker resilience as follows:

Definition 6.1 (Attacker Resilience) *The attacker resilience is a number giving the probability that a new established key between two arbitrary devices is secret under the assumption that the attacker was able to compromise c arbitrary devices.*

We denote the attacker resilience with $res(c)$. An attacker resilience of 100% means that we can guarantee the secrecy of any new key and an attacker resiliency of 0% means that the network is completely compromised. In order to provide graceful degradation the attacker resilience should be as high as possible for any number c of compromised devices.

Message Overhead *pack*

One of the most important performance metrics for any protocol in wireless sensor networks is the message overhead of a protocol. This is due to the fact, that the message overhead correlates directly with the used energy on any single node. Since energy is a very scarce resource on sensor nodes which are usually battery powered, any protocol for wireless sensor networks should use as few as possible transmitted messages. We evaluate the message overhead by analyzing the total number of packets sent in the network.

Memory Consumption *mem*

Even more crucial than the message overhead is the memory used (RAM) on each device. Any algorithm requiring more memory than available on the devices is not usable at all. Ideally, the memory usage should not increase when increasing the total network size. We analyze the RAM usage by analyzing the memory consumption of our approach on a single node. Besides the RAM usage, the code footprint, i.e. the ROM (or flash) usage, of our approach is also of importance. We measure memory consumption in bytes.

6.2 Influencing System Parameters

The most important parameters which influence our performance metrics are the security level s (and closely related the redundancy level r), the total number of nodes in the network n, the number of compromised nodes c, and the graph structure. We give a short explanation of each parameter in the following.

Security Level s and Redundancy Level r

The parameters s and r, i.e. $s+r$, dictate the graph connectivity. We introduce the connectivity level of our key graph as $z := s+r$. For the basic and extended approach we set $r = 0$. The connectivity level z directly influences the memory overhead and the memory consumption. Besides that, the security level s as defined in Section 2.4 directly influences a network's resilience against attackers, i.e. $a_{res}(c) = 100\%$ for all $c < s$.

Network Size n

Ideally, the network size n would not influence any performance metric given before at all. As we will see, memory consumption and processor usage are complectly independent of n. However, the message overhead depends on the network size.

Number of Compromised Nodes c

We denote with c the number of nodes an attacker was able to subvert. The attacker is able to choose nodes freely. The number of compromised nodes clearly influences the attacker resilience.

Graph Structure

In Section 5.6 we introduced three different graph structures, namely random, k-degree and chain. We perform our analysis on each of these graph types in order to gain a better understanding of the influence of how the choices made when adding a new node to the graph will later influence the network's performance.

6.3 Analysis

As shown in the previous chapters, our three key establishment algorithms together with the key graph construction rules guarantees that key establishment is possible for any two devices in the network. To analyze our key establishment protocol we now discuss the theoretical worst cases with respect to attacker resilience, memory usage, and network traffic.

6.3.1 Attacker Resilience

In this section, we provide an security analysis of our key distribution schemes. To do so we calculate the attacker resilience *res* of our network as defined in Section 6.1. First we develop a formula to compute the attacker resilience that is independent of the used key distribution scheme. After that we derive specific formulas for the basic and the fault-tolerant approaches, as well as for the extended approach.

All our approaches use s paths to establish a new key between two arbitrary devices. We define the length of these paths as the number of intermediary nodes on a path. For the basic and the fault-tolerant approach, the length of paths depends heavily on the actual graph structure and may differ for each node pair trying to establish a new key. In our extended approach, paths have always length one. For our analytical evaluation we abstract from the actual length of each individual path and introduce the parameter p_{avg}, denoting the average path length in a given graph.

We assume that the attacker was able to subvert c randomly chosen devices. As shown in the previous chapters the attacker cannot compromise a new key if $c < s$. Hence, what remains to

be examined is the case $c \geq s$. In such cases, the attacker may be able to compromise a new key, if and only if there is at least one subverted node on each path. Intuitively, the probability for this will rise with higher c. To compute the probability more exactly, we use the well-known hypergeometric distribution (e.g. [Bos95]). This distribution is a discrete probability distribution describing the number of successes S in a sequence of C draws from a finite population N without replacement. A draw is defined to be successful if the drawn object belongs to a special set F with $F \subset N$. We call F the set of favorable objects. The hypergeometric distribution is computed using the probability mass function f_{pm} given as:

$$f_{pm}(S,C,F,N) := \frac{\binom{F}{S}\binom{N-F}{C-S}}{\binom{N}{C}} \tag{6.1}$$

Equation 6.1 gives the probability for *exactly* S successes. However, for our analysis we are interested in calculating the probability of *at least* S successes. This corresponds to the cumulative probability which can be calculated with the cumulative distribution function (f_{cd}). Since the hypergeometric distribution is a discrete probability distribution the cumulative probability can be calculated by adding all corresponding single probability values:

$$f_{cd}(S \leq C,C,F,N) := \sum_{i=S}^{C} \frac{\binom{F}{i}\binom{N-F}{C-i}}{\binom{N}{C}} \tag{6.2}$$

With these generic formulas we can now develop specific formulas for our different key distribution schemes.

Basic and Fault-Tolerant Approaches

To compute the attacker resilience for the basic and the fault-tolerant approaches, we assign the variables S, C, F and N in Equation 6.2 as:

$$S = 1, \quad C = \binom{c}{s}, \quad F = (p_{avg})^s, \quad N = \binom{n}{s} \tag{6.3}$$

This is the case, as in total there are $\binom{n}{s}$ combinations of s-sized sets in a network consisting of n nodes, leading to $N = \binom{n}{s}$. If the attacker compromises c nodes he gains access to $\binom{c}{s}$ of s-sized sets, thus $C = \binom{c}{s}$. Furthermore, with s paths and each containing on average p_{avg} devices, there are $(p_{avg})^s$ favorable sets of s devices for the attacker ($F = (p_{avg})^s$). In order to compromise a new key the attacker needs only one correct s-sized combination, therefore $S = 1$.

With these assignments of the variables we can give the probability that an attacker is able to compromise a new key after subverting $c \geq s$ devices as follows:

$$f_{cd}(1 \leq \binom{c}{s}, \binom{c}{s}, (p_{avg})^s, \binom{n}{s}) = \sum_{i=1}^{\binom{c}{s}} \frac{\binom{(p_{avg})^s}{i}\binom{\binom{n}{s}-(p_{avg})^s}{\binom{c}{s}-i}}{\binom{\binom{n}{s}}{\binom{c}{s}}} \tag{6.4}$$

Using this probability, the attacker resilience for our basic and fault-tolerant approach is given as:

$$res_{bas} = \begin{cases} 1 & \text{if } c < s; \\ 1 - f_{cd}(1 \leq \binom{c}{s}, \binom{c}{s}, (p_{avg})^s, \binom{n}{s}) & \text{otherwise.} \end{cases} \tag{6.5}$$

Extended Approach

In the extended approach (see Chapter 5), keys are established by recursively establishing new keys between the source and the target of a key establishment request until both have keys to s direct key graph neighbors of the target. The final key is established via these neighbors. Therefore, to compromise the key, the attacker needs to subvert the s direct neighbors of the target used for the last key establishment, i.e. the number of favorable devices is exactly s (only compromising the s direct intermediaries between source and target would lead to the disclosure of the new key). Furthermore, since there are only s favorable devices, there is no need for the cumulative probability. Instead, we use the equation for the probability mass function from Equation 6.1 with $S = s$, $C = c$, $F = s$ and $N = n$:

$$f_{pm}(s,c,s,n) = \frac{\binom{s}{s}\binom{n-s}{c-s}}{\binom{n}{c}} = \frac{\binom{n-s}{c-s}}{\binom{n}{c}} = \ldots = \frac{\binom{c}{s}}{\binom{n}{s}} \tag{6.6}$$

Informally, Equation 6.6 can be explained as follows: There are a total of $\binom{n}{s}$ combinations of s-sized sets, while only one of these sets is favorable for the attacker. Hence, the probability to pick exactly the right combination is $\frac{1}{\binom{n}{s}}$ when picking exactly s devices. However, the attacker picks $c \geq s$ devices. Within the picked c devices, there are $\binom{c}{s}$ combinations of s-sized sets. Therefore, when compromising $c \geq s$ devices, the attacker has $\binom{c}{s}$ possibilities of having the right combination among the subverted devices. Thus, for the extended approach the attacker has a $\frac{\binom{c}{s}}{\binom{n}{s}}$ chance to gain access to a newly established key. This leads to a attacker resilience of the network as follows:

$$res_{ext} = \begin{cases} 1 & \text{if } c < s; \\ 1 - \frac{\binom{c}{s}}{\binom{n}{s}} & \text{otherwise.} \end{cases} \tag{6.7}$$

Discussion

Using the formulas given in Equation 6.5 and Equation 6.7 we can now plot the attacker resiliences res_{bas} and res_{ext} in dependence of the number of subverted devices and different s. The results can be seen in Figure 6.1. We used $n = 100$ and $p_{avg} = 4$, and varied s between one and ten and c in the interval $[0 \ldots n]$. Note that the basic and the fault-tolerant approaches are combined in the figure, as both have the same attacker resilience.

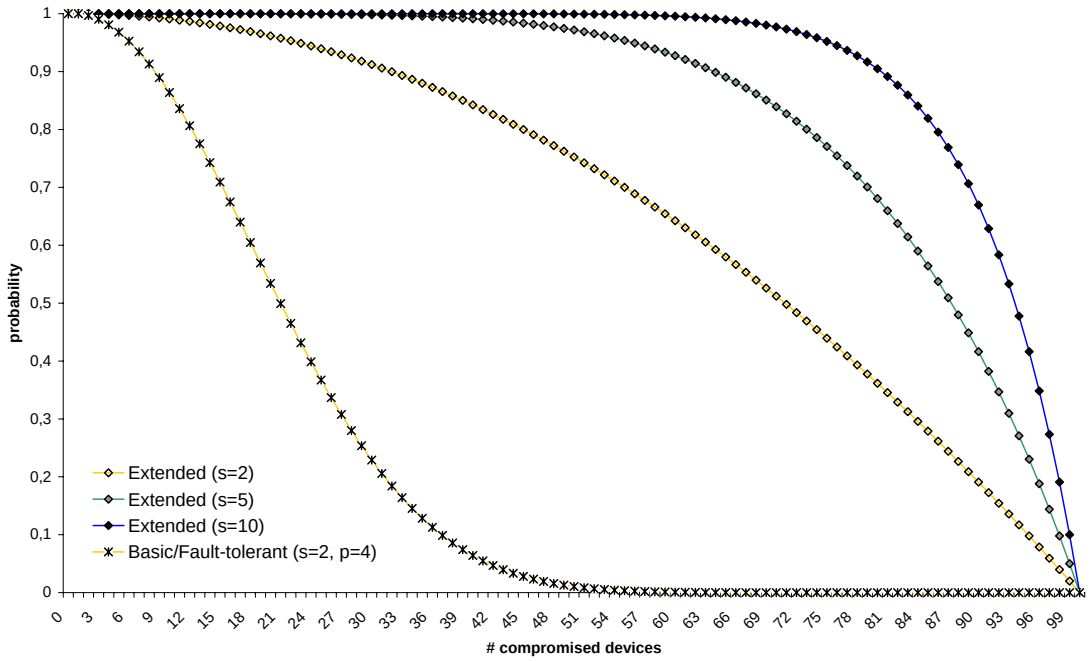


Figure 6.1: Attacker resilience for $s = \{2, 5, 10\}$, $n = 100$, $p_{avg} = 4$

As expected the extended approach shows an higher attacker resilience than the basic and the fault-tolerant approaches. This is simply due to the fact, that with the extended approach, there are less favorable possibilities available for the attacker. In addition, the attacker resilience grows with higher s for all approaches. However, the extended approach is able to profit more from higher s. Thus, with increasing s the difference between the extended and the other two approaches becomes increasingly prominent.

6.3.2 Memory Usage

In this section, we analyze the additional memory (RAM) needed on each device. We distinguish between network setup, i.e. the memory needed for the initial key graph, and key establishment, i.e. the memory needed when two devices establish a new key.

Furthermore, we can distinguish the memory consumption used for storing local algorithm variables like e.g. state and the memory consumption used for storing keys, i.e. the two key stores (see Section 2.5). Since the memory usage for the local algorithm variables are highly implementation dependent and also constant, we focus in our analysis on the memory consumption of the key stores, i.e. the amount of keys which need to be stored on a single device.

Network Setup

Due to our key graph construction algorithms (Algorithm 3.1, Algorithm 4.1 and 5.1) each device needs z (with $z = s + r$) shared keys to connect to the initial key graph. Since each key is stored on two devices, we have an average of $2z$ keys per device, which is independent of the total network size n. Hence, we require only a constant amount of memory on each device, i.e. the initial key graph requires on average $O(1)$ memory space on each device.

Assuming a key length of l_k bit, we calculate the average memory consumption in bytes on each device as follows:

$$mem_{avg} = 2z\frac{l_k}{8} = 2(s+r)\frac{l_k}{8} \tag{6.8}$$

Clearly, the actual memory requirement may be higher for some devices. This largely depends on the structure of the key graph (see Section 5.6 for a discussion of different structures). The worst case with respect to memory consumption is a "star" shaped key graph where each new device after the first s devices shares a key with the first s devices. Thus, the first s devices would need to store $n-1$ keys and would require $O(n)$ space. However, such graphs can be avoided at construction time by controlling the introduction of a new device. One way to do so is to limit the total number of keys stored in the key store by a parameter $d_{max} \geq 2z$, resulting in a k-degree key graph structure. In this case the maximum memory consumption can be calculated as:

$$mem_{max} = d_{max}\frac{l_k}{8} \tag{6.9}$$

Finally, in the special case of a "chain" key graph structure we have exactly $d_{max} = 2z$. Hence, the memory consumption on each individual node will not exceed $mem_{max} = 2z\frac{l_k}{8}$ bytes.

In conclusion we can say, that if we avoid some graph structures by controlling the introduction of new devices, our key graph algorithms use only a constant amount of memory on each device for the initial key graph, i.e. memory consumption is in $O(1)$-space.

Key Establishment

Our algorithms need additional memory on some devices when establishing a new key. With the basic and the fault-tolerant key establishment algorithm the two devices that establish a new key need only to store the new key shares, i.e. they need space for one additional key. All intermediary nodes need to store and forward the key shares, hence they also need only space for one additional key share. However, these two algorithms need to discover the s device-disjoint paths using a path search algorithm (see Section 3.5). Hence, the total memory consumption for a key establishment is heavily dependant on the used path search algorithm and its specific implementation. We measure the total memory usage for a key establishment using a max-flow algorithm in Section 6.4.3.

For the extended key establishment algorithm each entry in the *QuerySet* of a device (one per query) consists of the *RequestingSet* and *EstablishedSet*, which, taken together, contain at most the direct neighbors of this device, i.e. on average $2z$. Moreover, for each query, at most one temporary key will be established between a device and the target device. Thus, a query requires $O(1)$ space on each intermediate device and at most $O(n)$ on the target device. The exact memory requirement on the target device depends on the key graph structure: Every device that receives a query tries to establish a new key to the target device. When it has done so, it sends a "Cancel"-message notifying other devices that their temporary keys are no longer needed. Thus, at least s temporary keys need to be held on the target device in order to establish a new one, i.e. in total $s+1$.

However, there is not only one device trying to establish a new key to the target, the amount of temporary keys on the target device depends on how many devices do simultaneously establish a key to the target device. The number of such devices is highly influenced by the key graph structure. If we have a "chain" graph structure the number of additionally temporarily needed keys on the target device is exactly $(s+1)$, since there is always only one parallel temporarily key establishment. In contrast, when using a k-degree key graph structure the memory requirements on the target device is $k(s+1)$ with k being constant. This is because every parallel running temporary key establishment needs $s+1$ additional keys. In our emulation we examine the maximum temporary memory required on the target device experimentally.

In conclusion, the memory needed on each device for the basic and the fault tolerant approach depends on the path search algorithm. For the extended key establishment algorithm the memory needed is independent of the total network size n, hence each device needs only a constant amount of memory.

6.3.3 Network Traffic

In this section we analyze the message overhead of our key distribution schemes. To do so, we count the number of packets sent for network setup and key establishment respectively.

In our key distribution schemes messages are usually sent over encrypted connections, i.e. over edges in our key graph $G_K = (V, E_K)$. Hence, the total number of edges in our key graphs is of utter importance and can be calculated for a given $|V| = n$ as follows:

$$|E_K| = \begin{cases} \frac{n(n-1)}{2} & \text{if } n \leq s; \\ (n-s)s + \frac{s(s-1)}{2} & \text{if } n > s. \end{cases} \tag{6.10}$$

The correctness of this equation can easily be seen: As long as the key graph consists of fewer than s nodes, we have a fully connected key graph. In a fully connected key graph with n nodes each node shares an edge with $n-1$ other nodes. With the addition that each edge belongs to two nodes we get the upper part of the above equation. If there are more than s nodes in our key graph, we can simply count the number of edges as they are introduced: For each node introduced we also introduce s edges. Hence, we get $s(n-s)$ edges for all nodes introduced after s nodes are already in the graph, and – as already seen – the first s nodes lead to $\frac{s(s-1)}{2}$ edges leading to the lower part of the equation.

Network Setup

When introducing a new device into the network with our basic or fault-tolerant key distribution schemes no messages are sent over the wireless channel. Hence there is no overhead at all.

With our extended approach the amount of exchanged packets depends on the used method for introducing a device. When using the *configuration-device* or the *manual physical contact* method (see Section 5.5), only neighbor lists are sent over the wireless channel. Since the size of the neighbor lists is constant, a constant number of messages is transmitted for each new device. Thus, for the setup of the whole network we need $O(n)$ packets. In contrast, when using the *automatic physical contact* method, we need at most $(s-1)$ new key establishments, whereas each key establishment is in $O(n)$ (see next paragraph for explanation). Thus, the number of messages needed for the network setup is $n(s-1)O(n)$, i.e. $O(n^2)$.

Key Establishment

When two devices want to establish a new key with the basic approach, they split up the new key in s parts and send these paths over s device-disjoint paths to the target device. Hence the

amount of needed packets is the sum of the packets needed for the path search and the packets needed for transporting the key shares.

Analogously to the memory consumption, the packets needed for the path search depend heavily on the algorithm used and its actual implementation. Hence, in our analysis we focus on the packets needed for the key share transportation. Assuming again that we have an average number of p_{avg} of intermediary nodes (i.e. path length) the number of packets needed for transporting the key shares is as follows:

$$pack_b = s(p_{avg} + 1) \tag{6.11}$$

The amount of packets needed for the fault-tolerant approach is equal to the number of packets needed for the basic approach if there was no intruder disturbing the protocol. If an attacker disturbs the protocol, the complete protocol may have to be restarted. Hence, if the attacker interferes with the protocol i times, we can calculate the message overhead as follows:

$$pack_{ft} = (i+1) \cdot pack_b = (i+1) \cdot s(p_{avg} + 1) \tag{6.12}$$

With the extended key establishment algorithm, the total amount of messages for one query is linear in n, i.e. $O(n)$: Each "KeyEstablishmentQuery" is sent at most twice on each key edge, each "Established"-message is sent at most once per key edge and each "Cancel"-message is also sent at most once per key edge. Furthermore, a "KeyValid"-message is sent at most n times, i.e. by each device once in the worst case where every device establishes a temporary key to the destination device. Since in the extended key establishment algorithm we use paths with exactly one intermediary node, we get at most $2n$ messages for transporting key shares. Thus, we get at most $4|E_K| + 3|V|$ messages for a key establishment. Hence, with Equation 6.10 we get the total amount of packets for the extended approach as follows:

$$pack_{ext} = 4|E_K| + 3|V| = 4 \cdot [(n-s)s + \frac{s(s-1)}{2}] + 3 \cdot n = (4s+3) \cdot n - 2s(s+1) \tag{6.13}$$

As an example, if we use a security level of $s = 2$ we would get in the worst case about $pack_{ext} = 11n - 12$, hence for $n >> s$ we can estimate $pack_{ext} \approx 11n$.

6.3.4 Summary

In this section we have analyzed the attacker resilience, memory requirements and network traffic of our three key establishment schemes analytically. Our analysis shows, that the extended approach outperforms the basic and the fault tolerant approach in terms of the resilience

against attackers. In terms of memory usage per node, all approaches have a complexity of $O(1)$ if we neglect the memory needed by the path search algorithm in the basic and the fault tolerant approach. The network traffic for the basic and the fault tolerant approaches depend on the average path length p_{avg} between nodes establishing a new key and – again – the used path search algorithm. The extended approach needs $O(n)$ messages to establish a new key.

6.4 Emulation

In addition to our analytical evaluation, we performed a number of experiments to strengthen our results. In this section we present our measurements. We start by presenting our emulation environment in Section 6.4.1 and the used parameters in Section 6.4.2 respectively. The emulation results of memory consumption are then presented in Section 6.4.3 and the ones of message overhead in Section 6.4.4. We close this section with a short summary in Section 6.4.5.

6.4.1 Emulation Environment

To evaluate our approach, we have implemented the different key distribution schemes with ANSI C. Our implementation is designed to be compatible with highly resource-constrained devices, e.g. no threads etc. are used. Therefore, our software can run – for instance – on a Microchip PIC (e.g. used int the SmartIt platform [Sma]) or an Atmel CPU (e.g. used in the Berkley Motes platform [Mot]) without difficulty.

For our experiments we executed our system using the Network Emulation Toolkit (NET) [HMTR04, NET]. NET allows designers of network protocols to test their protocols by offering a toolkit to design and set up virtual networking scenarios. As an example, the user may design a scenario with 50 mobile nodes which are connected via IEEE 802.11 [CWKS97]. NET is based on a Linux cluster comprised of 64 personal computers which are connected over a Gigabit Ethernet network. In addition, we have built a management tool which enables the semi-automated execution of different test settings and collects the results.

6.4.2 Parameter Selection

For our experiments we used scenarios of 25 to 250 devices in increments of 25. We also varied the security level s to show how this influences the measured values. We measured the number of packets and the amount of memory used on a device. The memory measurements are based on the assumption of 128-bit symmetric keys and 16-bit device IDs.

Nodes are always in transmission range of each other and transmitted packets are never lost. That way, the measurements are not polluted with effects not directly related to the key establishment protocols.

To detect the needed node-disjoint paths, the basic and the fault-tolerant approaches need an additional path search algorithm. To detect these paths we need to solve the max-flow (MF) problem [ET75]. We chose this algorithm (see Section 3.5), as it is often considered in the literature (e.g. [Gol98]) to be one of the most efficient algorithms for the max-flow-min-cut problem. MF operates on a global view of the network on each device: Each device that is added to the network transmits a link-state packet containing all its neighbors. This packet is flooded through the network using the encrypted links. With a global view of the network, each device can locally search for the node-disjoint paths in the key graph using the MF algorithm. After the node-disjoint paths are found, the key shares are transmitted using source-routing.

6.4.3 Memory Usage

In this section we measure the memory usage of the basic and the extended approach. The results for the fault-tolerant approach resemble those for the basic approach closely and are therefore omitted.

Network Setup

We measured the maximum memory a single device needs while the network grows for $s = 2$ and $s = 3$. We performed the measurement for our basic approach utilizing the max-flow path search algorithm and for the extended approach. As discussed in our analytical discussion (see Section 6.3.2) the memory consumption for the extended approach depends on the used graph structure. Hence we performed measurement for a random and a chain key graph structure. The results of this measurement are presented in Figure 6.2.

As we can observe in the diagram, the memory consumption of the extended approach with $s = 2$ and $s = 3$ remains constant. This is inline with our analytical model for the memory consumption with a chain key graph structure, i.e. a k-degree key graph. When using a random key graph we can observer that the memory requirement grows about linearly with the number of devices.

On the other hand, when using the basic approach with the max-flow path search algorithm we notice a very high memory requirement. Even though, the memory needed for storing the keys is also here constant, the memory needed by the max-flow path search algorithm dominates the memory requirement. The basis for the MF algorithm is a global view of the entire graph and each device needs this view to determine the node-disjoint paths. For storing a graph

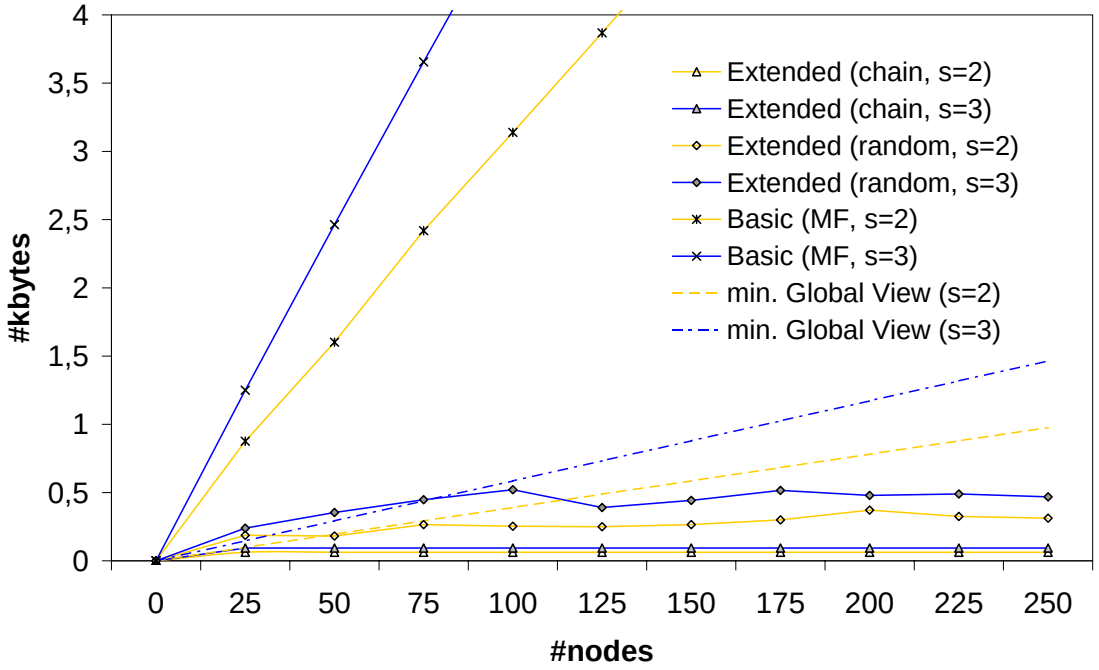


Figure 6.2: Maximum memory usage during setup

representing the global view, we need at least linear memory: With 2 bytes per device ID, the minimum amount needed for storing the whole graph would be $2ns$ bytes. For comparison, these amounts are also shown in the diagram as dotted lines. Hence, this is the lower limit of memory requirements when using the basic approach with MF. The actual memory requirement is highly implementation dependent. Our implementation shows a higher memory requirement due the used method for storing the key graph. However, there is still room for optimizations.

Key Establishment

We measured the maximum additional memory requirements during key establishment with $s = 2$ and $s = 3$. We performed the measurement for our basic approach utilizing the max-flow path search algorithm and for the extended approach. Also here we used for the emulation of the extended approach random and chain key graphs. The results of this emulation are presented in Figure 6.3.

The diagram shows a similar pattern as with the memory consumption during network setup: The extended key establishment algorithm uses only constant memory with the chain key graph structure, which is inline with our analytical model. In case a random key graph is used, the additional memory requirements grow very slowly in a linear fashion. Note that the linear growth with a random key graph is due to the memory requirements on the target node. The

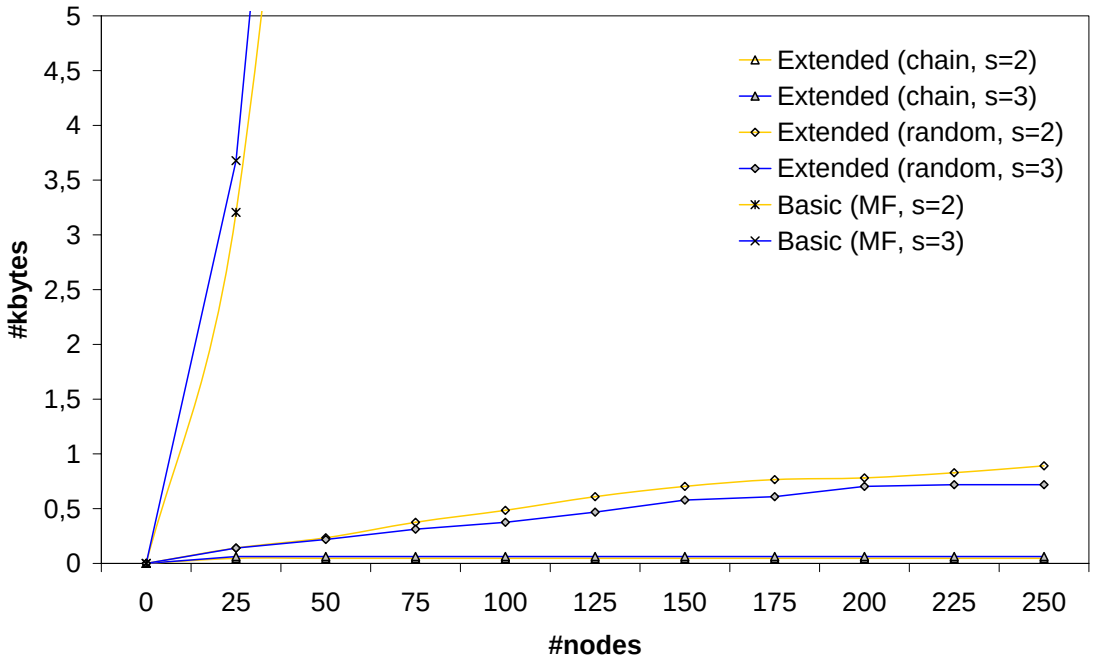


Figure 6.3: Maximum memory overhead per device during key establishment

extended key establishment algorithm has an interesting effect. The number of temporary keys that needs to be established decreases for higher values of s. This is a result of the s-connectors growing larger, hence with the same amount of total nodes only fewer temporary keys are possible.

The memory consumption of the basic approach with the max-flow is very high for the source node (which has to compute the node-disjoint paths). For this node, memory consumption grows very fast. This is due to the fact that MF algorithms work on directed graphs and find *edge*-disjoint paths: In order to be used for our purposes, a graph transformation is needed that generates a graph of a size proportional to the square of the original graph size. This graph transformation is very memory-consuming, hence the enormous high memory overhead with the basic approach. Since the memory overhead for the basic approach is dominated by the memory needed for the graph transformation, there is still room for improvement.

6.4.4 Message Overhead

In this section we present our emulation results with regard to message overhead of our basic and extended key distribution scheme. We measure the total number of packets sent during network setup and key establishment. Note that the key graph structure does not affect the message overhead caused by our extended approach: For network setup, only a constant number of

packets are needed to exchange the initial keys and for key exchange. For key establishment, all messages will be sent over all edges, regardless of the structure of the graph. Hence, in the following emulations we do not distinguish between different graph structures.

Network Setup

In this experiment we counted the total number of packets sent when setting up the network, i.e. packets needed for incrementally building the key graph by adding nodes one to n. We performed our experiment for the basic approach utilizing the MF algorithm and the extended approach using the configuration device method (see Section 5.5). In both cases we used $s = 2$, $s = 4$ and $s = 6$ as our security level. The cumulated results for these experiments are presented in Figure 6.4.

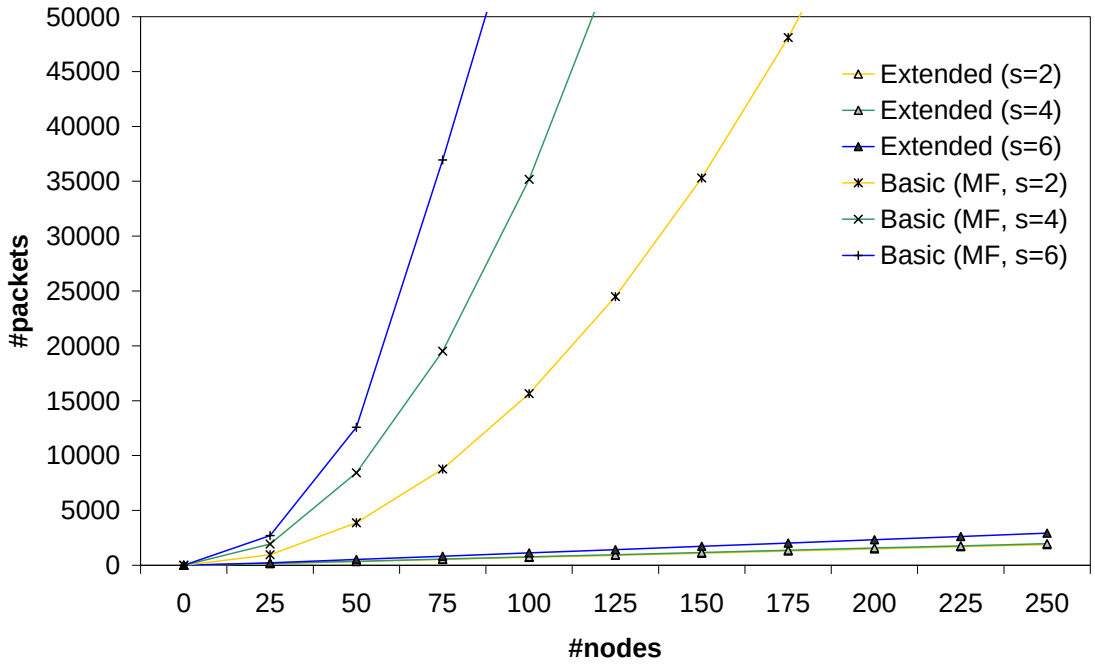


Figure 6.4: Total messages during network setup

We can see a very slow linear growth in the messages needed when using the extended approach. This can easily be seen: Introducing a new device into the network requires a constant number of packets to communicate the neighbor-lists to the configuration device and another ks messages to store the first s new keys of the new device on already existing devices. Hence, when each device needs a constant number of packets, introducing n devices needs $O(n)$ packets leading to the linear growth. Note, when using e.g. the manual physical contact or the static

pre-distribution method there would be no message overhead at all when using the extended approach.

The basic approach using the MF algorithm as the path search algorithm shows a quadratic growth. The reason for this is that whenever a single device is added to the network, the global view which is stored on every device needs to be updated. Hence, a new device which is introduced in the network must communicate its presence to all existing devices. This is done most effectively by flooding the network. Hence, introducing a single device leads to a linear overhead of messages. Therefore for introducing n devices n-times linear overhead is needed, hence quadratic growth.

Key Establishment

In this experiment we evaluated the message overhead produced by our basic and our extended key distribution scheme during key establishment. We counted the total number of packets sent during the establishment of a new key between two arbitrary devices. Again we chose a security level of $s = 2$, $s = 4$ and $s = 6$. However, the security level s influences only the extended key establishment algorithm while for the basic key establishment algorithm the difference between different s-values is negligible. Furthermore, we used Equation 6.13 to calculate the number of packets sent. The analytical and simulated results are presented in Figure 6.5.

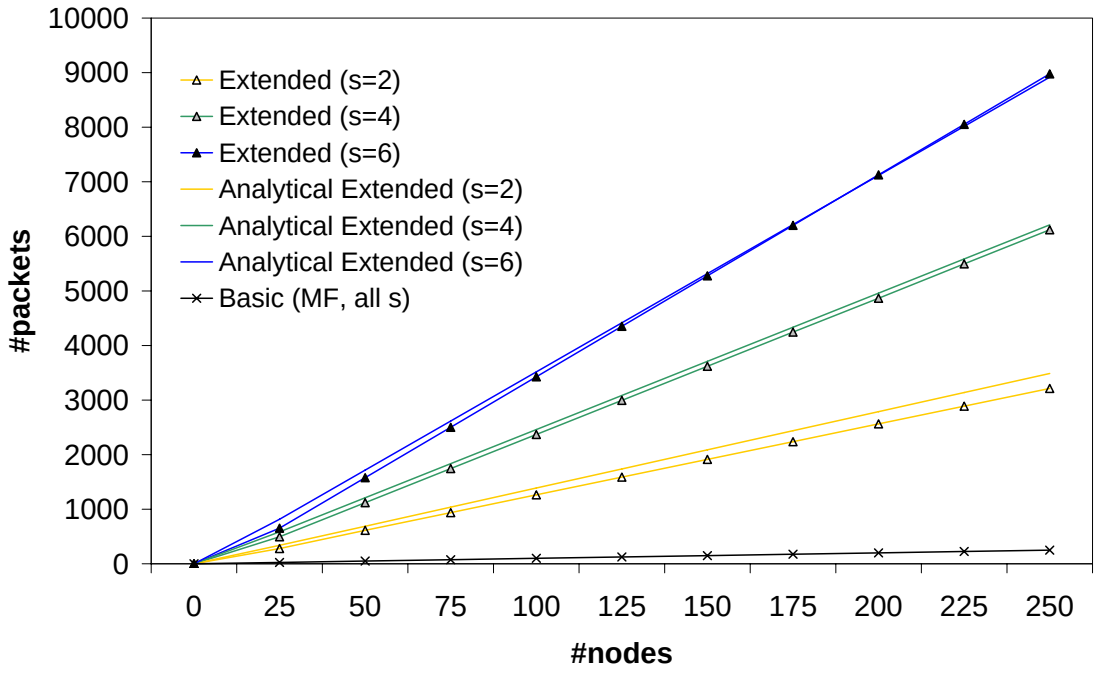


Figure 6.5: Total messages during key establishment

In this scenario, the basic approach using the MF algorithm for path search clearly uses less messages than the extended key establishment algorithm. The basic approach can trade off memory for network traffic: The only packets needed for key establishment are those that transport the actual key shares. The plotted line representing the basic approach shows the worst-case effort needed to do this.

The extended approach requires more messages, but still manages a linear growth rate with respect to the number of devices. This is simply due to the flooding nature of our extended approach, i.e. messages must reach any device in the network in order to find the right one.

6.4.5 Summary

In this section we complemented our analytical results from previous sections with experimental results. On the one hand, these results are inline with our analytical model. On the other hand we gained more precise insight in the dependence of our algorithms from the path search algorithm. With these results in mind it is now possible to discuss in which scenario which algorithm should be used.

6.5 Discussion

We provided in this chapter an evaluation of our key distribution schemes. We can conclude that our extended approach is always better with respect to the attacker resilience. Furthermore, the memory requirements are constant in the case of a k-degree key graph structure and sublinear otherwise. Thus, from the point of memory footprint this algorithm can scale infinitely. In contrast, the network traffic during key establishment grows linearly and therefore limits scalability. However, this is a general problem of a reactive algorithm that does not use any knowledge about the network: The only way to ”find” another device is to query all other devices, i.e. communicate with $(n-1)$ devices. Thus, for a reactive algorithm with no additional information linear growth is also the lower bound. Therefore, the extended key distribution scheme represents a good trade-off between memory requirements and network traffic. We achieve constant memory requirements while causing linear growth in network traffic, which is the lower bound for reactive algorithms.

Analyzing the memory overhead and the message overhead of our basic and fault-tolerant approach, we found that the approaches are slightly better than the extended approach. However, as showed in our simulated experiments, this effect is countered and completely eradicated when taking the path search algorithm into account. Hence, the overhead of the basic and fault-tolerant approach is completely dominated by the path search algorithm.

Analyzing our experimental results with respect to the path search algorithm we can conclude that global view algorithms can be used in small networks, since the memory overhead for the global view quickly grows larger than the available memory on resource-constrained devices. Therefore, in large networks global view algorithms should be avoided. As already discussed in Section 3.5 the distributed algorithms for path search are a good alternative. However, the best known distributed algorithm by Ogier and Shacham [ORS93] uses logarithmic memory and quadratic communication overhead in order to find node-disjoint paths, which is still a higher overhead than our extended key distribution algorithm needs.

In conclusion, our extended key distribution scheme outperforms our basic and fault-tolerant approach in all performance metrics. as showed in our analysis, the attacker resilience of our extended approach is always better. With respect to memory and communication overhead the basic and fault-tolerant approach is only as good as the path search method used. However, there is no path search algorithm available which uses less memory and/or communication overhead than our extended approach.

7

Related Work

In this chapter we present existing key distribution schemes and compare them to our approach. The chapter is structured as follows: First we discuss key distribution schemes in other application areas, namely classical internet-based schemes in Section 7.1 and key distribution schemes for mobile ad hoc networks in Section 7.2. After that we focus more closely on key distribution in wireless sensor networks (Section 7.3). Finally, we give a summary of this chapter including a tabular overview of all existing approaches with respect to our requirements as presented in Section 2.2. This comparison once again shows the novelty of our approach.

7.1 Classical Key Distribution Schemes

In the early days of cryptography, communication partners agreed on a set of algorithms to en-/decrypt messages. These algorithms were known only to these communication partners, i.e. the common secret between the partners were the cryptographic algorithms used. In modern cryptographic systems, the used algorithms are well known to all participants of the system. Instead of a set of algorithms, communication partners exchange keys that are used in the algorithms.

The earliest approach for a key exchange scheme is the *Diffie-Hellman* key exchange algorithm [DH76] used e.g. by *Secure Socket Layer* (SSL) [FKK96], its successor *Transport Layer Security* (TLS) [DA99], and the Internet Key Exchange (IKE) protocol [HC98]. It allows two communication partners to securely establish a common symmetric key using asymmetric cryptographic operations, i.e. modular arithmetic. However, pure Diffie-Hellman is vulnerable to man-in-the-middle attacks, as the communication partners are not authenticated. To solve this problem, the communication partners can sign their messages [DvOW92]. This requires a certification infrastructure, e.g. based on the X.509 ITU-T recommendation [X.597].

Another well known approach to exchange keys is the Kerberos system [KN93]. Kerberos is a variation of the Needham-Schroeder algorithm [NS78] which uses a central trusted authority to exchange keys securely and reliably. In contrast to Diffie-Hellman, Kerberos relies on symmetric cryptography only.

Classical key distribution schemes do not fulfill our requirements concerning decentralized operation (Section 2.2) and reliance on symmetric cryptography (Section 2.2) and thus cannot be used in wireless sensor networks with highly resource-constrained devices.

7.2 Key Distribution Schemes in Mobile Ad Hoc Networks

Mobile ad hoc networks consist of a number of mobile devices that communicate with each other. The network topology changes frequently due to device mobility and temporary network partitions are common. Therefore, similar to wireless sensor networks, centralized solutions cannot be used in mobile ad hoc networks. Different decentralized key distribution schemes have been developed for them (e.g. [ZH99], [HBC01]).

One such approach is given by Zhou and Haas in [ZH99]. They propose to select an arbitrary subset of n devices from the mobile ad hoc network and to distribute the functionality of a centralized certification authority between them. To do so they use *(n,k) threshold cryptography* [Sha79, DF89]. To create a new certificate, k members of the certification authority subset must sign it. This allows the system to operate reliably even if some devices are not available or subverted by an attacker.

Hubaux et al. propose a completely distributed certificate infrastructure based on certificate chains similar to PGP. To do so, each device stores a predefined subset of all certificates in a local repository. When two devices meet, they share their repositories and try to determine a certificate chain between them.

Although these schemes work in a decentralized way, they rely on asymmetric cryptography and thus do not fulfill all our requirements (see Section 2.2).

7.3 Key Distribution Schemes for Wireless Sensor Networks

Several key distribution schemes for securing wireless sensor and/or actuator networks have been proposed in the literature. The first key distribution scheme for wireless sensor networks was presented in 2001 by Perrig [PSW+01]. It workes with a central trusted authority, e.g. a trusted base station. In 2002 the first random key distribution scheme by Eschenaur and Gligor [EG02] opened a new domain of research: the decentralized key distribution schemes for

wireless sensor networks. However, due to the probabilistic nature of this approach, it cannot guarantee that any two devices in a sensor network can establish a new key. The first distributed approach that was able to provide this guarantee was proposed by Liu et al. in 2003 [LN03]. However, this approach is based on polynomial computations and is therefore not suitable for highly resource-restricted devices. Our key distribution schemes represent the first ones using symmetric cryptography only and do not use a random pre-distribution of keys, hence being applicable for highly resource-constrained devices and *guaranteeing* key establishment between any two devices.

In the following, we give an overview of existing key distribution schemes for wireless sensor networks. To do so, we structure these approaches according to Figure 7.1.

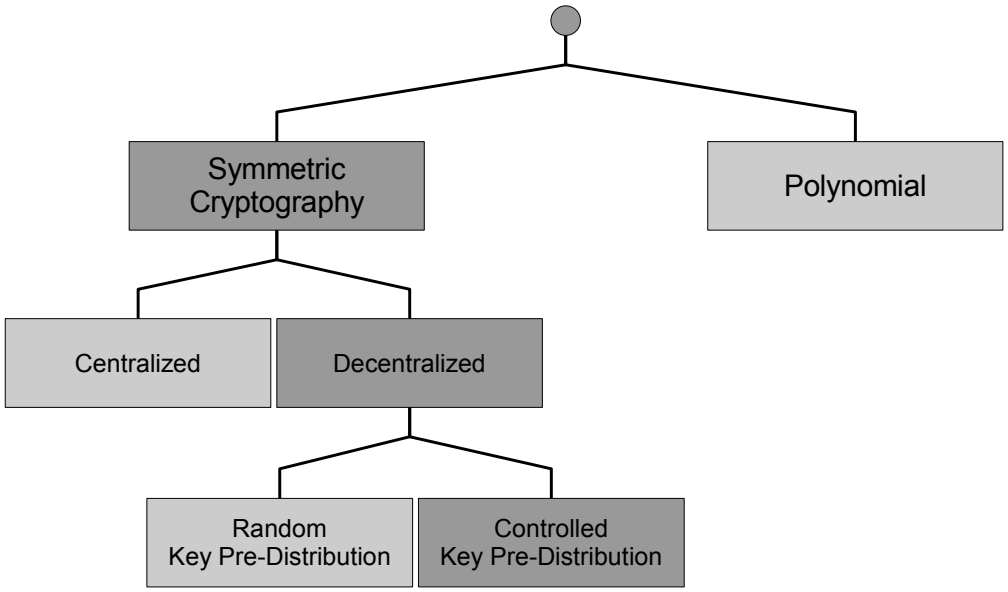


Figure 7.1: Classification

First, we discuss key distribution schemes using symmetric cryptography. For these approaches we distinguish schemes that have a centralized system organization (Section 7.3.1) and schemes that operate in a decentralized way (Section 7.3.2). We further divide decentralized schemes into approaches using random key pre-distribution and approaches using controlled key pre-distribution. After that we focus briefly on polynomial-based key distribution schemes in Section 7.3.3.

7.3.1 Centralized Key Distribution Using Symmetric Cryptography

Due to the high resource consumption of approaches involving asymmetric cryptography, the use of symmetric cryptography is proposed in many recent publications, e.g. [PSW+01,KSW04,

CP05]. The simplest and most straightforward solution is the usage of a resource rich key distribution center (KDC). These approaches are therefore also referred to as KDC-based approaches.

As an example, SPINS [PSW+01] provides a KDC-based key distribution approach. Each sensor device shares a symmetric key with the KDC. To establish a new key with another sensor device, the KDC is contacted and instructed to generate the new key and distribute it to the communication partners.

The main advantage of KDC-based approaches is their simplicity and the fact that the resources used on each individual node are very low. Each individual sensor device only needs to secure its communication with the KDC. Resource intensive tasks, e.g. managing all keys available in the network, are executed by the KDC, which is considered resource-rich.

The main drawback of these approaches is that the KDC represents a single point of trust. If the KDC is subverted by an attacker, the whole network is compromised. In addition, the KDC is a single point of failure, hindering new key establishments in case of a KDC crash or disconnection. Lastly, as all sensor devices need to communicate with the KDC during key establishment, the devices around the KDC may experience a high communication amount in a large multi-hop network. This may deplete these devices' batteries quickly, shortening the network lifetime. The highly focused communication pattern also allows an adversary to easily perform traffic analysis to locate the base station for compromise.

7.3.2 Decentralized Key Distribution Using Symmetric Cryptography

To overcome the drawbacks of a central authority, recent work uses symmetric cryptography in a purely decentralized fashion, i.e. without the need for any central authority. The simplest and most straightforward approach is to use only a single symmetric key for all devices, e.g. like proposed by Basagni et al. in [BHBR01]. With this approach only group-authentication is possible and an attacker can subvert the whole network after subverting a single device. In [BHBR01] this problem is circumvented by requiring tamper-proof devices. We do not assume tamper-resistance for devices in our system model as discussed in Section 2.1.

Another simple and straightforward but memory-intensive scheme is the usage of a fully connected key graph as discussed, e.g. in [EG02, CP05]. In this scheme, each node in a network of n nodes shares a unique pairwise key with every other node in the network. The memory overhead is $n-1$ cryptographic keys for every sensor node, which again can be considered problematic for large networks with highly resource-constrained devices.

In order to overcome these problems schemes for pairwise keys have been developed. Here, only a subset of all keys is stored on each device. These schemes – to which also our key

distribution scheme belongs – can further be distinguished between random key pre-distribution and controlled pre-distribution. In the following we present existing solutions based on each of these schemes.

Random Key Pre-Distribution

Eschenauer and Gligor [EG02] proposed the first solution using random key pre-distribution. Before deployment, every device is supplied with a random set of keys from a key pool. After deployment, the devices try to establish connections by finding a commonly shared key or by creating a new key through a secure path including other devices. This approach can be divided in three phases. In the first phase (key pre-distribution phase) each sensor node randomly selects m distinct keys from a large pool of $|S|$ keys. After that, in the second phase (shared key discovery phase) each sensor node discovers all neighboring nodes which share at least one common key with it. If such a key exists, it is used to secure the link and these two nodes are said to be connected. After this key discovery phase a connected key graph should be formed. In the third phase (path key establishment phase), new keys can be established between two nodes that are previously unconnected in the key graph. To do so, the node that wants to initiate the secure communication selects a single multi-hop path in the key graph leading to its intended communication partner and sends a key over this path. At each hop of the path the key is encrypted using the shared symmetric key of the corresponding nodes. This approach has three main disadvantages. First, if a single node which is located on the chosen path is compromised by an attacker, the new key is compromised, too. Second, since the approach is probabilistic, no clear assumptions about key graph connectivity can be made, thus there is no guarantee that a suitable path can be found and thus that a key can be established. Third, keys are shared by more than two devices, hence no authentication can be provided.

Chan et al. [CPS03] and Zhu et al. [ZXSJ03] also use initially distributed sets of random keys. However, in addition to Eschenauer and Gilgor's approach, they provide authenticated links, i.e. a unique key is shared by exactly two devices. In addition, they use *multiple* paths for the establishment of new keys by splitting the pairwise key over multiple untrusted paths. This enhances the resistance against attackers considerably. However, due to the random key predistribution, the actual existence of different paths in the network is not assured in any way. Therefore, in contrast to our approach, no presumption about the real number of the device-disjoint paths is possible.

Du et al. [DDH+04] propose a random key pre-distribution method to reduce the memory usage of random key pre-distribution schemes using knowledge about the future deployment of the sensor devices, e.g. their approximate relative location. The basic idea is to store corresponding keys on devices that are expected to be located nearby after deployment, therefore reducing the number of needed keys for achieving a given level of connectivity. However, such knowledge

about the future deployment of sensor devices is not always available and not assumed in our system model.

All approaches presented in this section assume that networks can be pre-configured and that it is a priori known how big the network is, or might get. These approaches do also not address the issue of easy addition or removal of devices. In addition, the approaches cannot guarantee that the resulting key graph is connected adequately for guaranteed key distribution between any two devices in the network.

Controlled Key Pre-Distribution

Our key distribution scheme belongs to the class of controlled key pre-distribution schemes. In contrast to random pool-based pre-distribution schemes the resulting key graph can be deterministically constructed, i.e controlled. With pool-based random key pre-distribution schemes the resulting key is connected only with a certain probability. Furthermore, these approaches tend to either use at least $O(n)$ memory or communication overhead.

Concurrently to the development and publishing of our extended key distribution scheme another controlled key pre-distribution scheme has been development and published in 2005: Chan's "Peer Intermediaries for Key Establishment" (PIKE) [CP05], which we describe in the following.

The basic idea behind PIKE is to construct the key graph in a way that for any two nodes from the network there always exists at least one intermediary node (i.e. a node that shares a key with both of them). If this is given, it can be guaranteed that any two nodes, which do not share a key yet, can establish a new key using the intermediary node.

The approach assumes that the maximum size of the network is known a priori. With this assumption in mind, the approach works as follows: let the maximum size of the network be n. For now we assume that n is a perfect square – later we will generalize the approach for arbitrary n.

Each node in the network has an ID of the form (x,y). On deployment (i.e. key pre-distribution) each node needs to store a unique key to all other nodes $(x,i)\ \forall i \in \{0,1,2,\ldots,\sqrt{n}-1\}$ and (j,y) $\forall j \in \{0,1,2,\ldots,\sqrt{n}-1\}$. Hence, each node stores $2(\sqrt{n}-1)$[1] keys and therefore the memory overhead is $O(\sqrt{n})$.

Later, if two nodes need to establish a new key they first identify the two nodes to which both of them share a key due to the pre-distribution. After that, one of these intermediaries is chosen and used to establish a new key. To do so the first node sends the new key to the intermediary encrypted with the shared key. The intermediary node then decrypts the new key,

[1] This can be reduced to $\sqrt{n}-1$

and re-encrypts it with its shared key with the target node. After that, the two nodes share a new key and the target node sends a message to the source in order to confirm the reception of the new key.

To clarify the ID-scheme of PIKE, we present an example with $n = 100$ taken from [CP05]. This example is depicted in Figure 7.2.

00	01	02	03	04	...	09
10	11	12	13	14	...	19
20	21	22	23	24	...	29
30	31	32	33	34	...	39
⋮	⋮	⋮	⋮	⋮	⋮	⋮
90	91	92	93	94	...	99

Figure 7.2: PIKE: Sample virtual ID space for 100 nodes

The node IDs are numbered from $0 \ldots 99$ and arranged in a square. By doing so the scheme can easily be imagined: Each node needs to share a pre-distributed key with every other node on the same row and column. Hence, if for instance node 14 and 91 need to establish a new key, they discover that node 94 and node 11 are intermediaries. With the aid of one of these intermediaries the new key can be established.

In case the network is not yet completely deployed, one needs to ensure that the nodes are deployed in their order. If this can be ensured, it can be guaranteed that at least one intermediary node is always found. This method also covers the general case if the network size is not a perfect square: For any network size n' not being a perfect square the next higher perfect square needs to be used as the network size n, i.e. $n = \lceil \sqrt{n'} \rceil^2$.

However, in this approach the secret is not split up, making the intermediary node an implicitly trusted device. Thus, the secrecy of a new key cannot be guaranteed after a single node has been compromised. In contrast our approach guarantees the secrecy of any new key for up to s compromised nodes.

7.3.3 Polynomial-based Key Distribution Schemes

The first polynomial-based key distribution scheme was proposed by Blundo et al. [BSH+93], inspired by [Blo85]. It was intended for group key distribution for conference systems. Based on Blundo's approach, Liu et al. [LN03, LNL05] proposed a general framework for polynomial

pool-based pairwise key pre-distribution schemes for wireless sensor networks. In the following we first explain the approach given by Blundo et al. before we discuss the extensions and modifications done by Liu et al.

Applying the special case of pairwise keys, Blundo's scheme works as follows: First, a server – e.g. operated by the manufacturer before deployment – randomly generates a t-degree bivarante polynomial over a finite field F_q of the form:

$$f(x,y) = \sum_{i,j=0}^{t} a_{ij} x^i y^j \wedge f(x,y) = f(y,x) \tag{7.1}$$

The number q must be prime and large enough to accommodate the intended cryptographic key.

In a second step, the manufacturer's server generates for each device with the unique ID i a so-called *polynomial share* $p_i(y) := f(i,y)$ and distributes these shares to the device. Hence, each device stores exactly one polynomial $p_i(y)$. After this step, the devices can be deployed in the field.

When two devices need to establish a new key, they just need to know the ID of each other. Let's assume that a device with ID A needs to establish a new key with a device with ID B. To do so, both devices calculate their polynomial share $p_i(y)$ for the other device's ID, i.e. B calculates $p_B(A)$ and A calculates $p_A(B)$. These calculations yield the same result due to:

$$p_B(A) = f(A,B) = f(B,A) = p_A(B) \tag{7.2}$$

No other sensor node (or attacker) can calculate this value without the knowledge of $f(x,y)$, $p_A(y)$ or $p_B(y)$. Blundo et al. show in [BSH+93] that the value remains perfectly secure as long as the attacker was not able to recover either $f(x,y)$ or at least $t+1$ polynomial shares, with t being the degree of the polynomial. Hence, the two devices can use the result of this calculation as their new shared secret, i.e. their new symmetric key.

However, this key distribution scheme cannot be applied directly to wireless sensor networks with highly resource-constrained devices. First, the storage cost for the polynomial grows quadratically with the network size (i.e. with the ID-size). Second, the computation of such a polynomial share is based on modular arithmetic, similar to asymmetric cryptography. Therefore, this approach is not applicable directly to wireless sensor networks.

A variation of Blundo's general scheme for key distribution specially tailored towards sensor networks was presented by Liu et al. [LN03, LNL05]. They proposed a general framework for polynomial pool-based pairwise key distribution based on a combination of the concept

of Blundo for polynomial key distribution and the random key pre-distribution concept of Eschenauer and Gligor [EG02]. Instead of deploying a set of keys from a key pool as in the key pre-distribution schemes, bivariant t-degree polynomials are used to compute the pairwise keys.

The approach works as follows: Initially, a setup server randomly generates a pool of s bivariante t-degree polynomials over a finite field F_q. Each sensor node picks a subset of s' polynomials from the pool and is assigned the IDs of these s' polynomials. After deployment the nodes exchange the IDs of the polynomials stored on them, in order to detect polynomials shared with their neighbors. If two nodes share the same polynomial, they are said to have the same secret and are therefore able to generate a pairwise communication key. If two nodes fail to establish a direct link, they initiate a key establishment. If the source node is able to find a path to the destination node, a pairwise key can be exchanged via the discovered secure path. Given the network is connected, there always exists a path between any two nodes. To retrieve the original t-degree polynomials, an attacker requires to capture at least $t+1$ nodes.

Liu et al. propose two different variants of this scheme. In the first variant, the sensor devices pick the polynomials to include into their initial subset randomly. This results in an approach similar to a random key pre-distribution with respect to the resulting graph connectivity. In the second variant, polynomials are chosen deterministically similar to the approach used by PIKE [CP05] as discussed before. This results in a controlled pre-distribution which is able to guarantee the necessary graph connectivity.

7.4 Summary

In this chapter we presented and classified existing approaches for key distribution in wireless sensor and/or actuator networks. However, none of the presented approaches can fulfill all of our requirements as presented in Section 2.2. To clarify this, we summarize all approaches with respect to our requirements in Figure 7.3. The first column – *Decentralized* – shows if the approach works in a fully decentralized fashion, i.e. no central authority is needed. With the second column – *Symmetric Cryptography* – we express the usability of the approach for highly resource-constrained devices. This implicates that the corresponding approach does not use any resource-intensive asymmetric cryptographic operation, i.e. modular arithmetic on very large numbers. The third column – *Authentication* – implies that the scheme provides for device-to-device authentication. In the case of symmetric cryptography, this is only the case if unique pairwise keys are used. *Dynamic Network Size* – the fourth column, indicates that there is no predetermined network size, and devices can be added or removed at will. With the *Scalability* column we show which approach scales for very large networks. Scalability can be limited by e.g. memory overhead or communication overhead. Wether an approach can cope with device

failures is shown in column six. Lastly, in column seven – *Guaranteed Operation* – we indicate if an approach is able to guarantee the key establishment of two arbitrary devices.

	Decentralized	Symmetric Cryptography	Authentication	Dynamic Network Size	Scalability	Copes with Device Failures	Guaranteed Operation
Diffie-Hellman key exchange [DH76]	✓			✓	✓	✓	✓
Diffie-Hellman with X.509 [X.597]	✓		✓	✓	✓	✓	✓
Needham-Schroeder [NS78]		✓	✓	✓	✓	✓	✓
Kerberos [KN93]		✓	✓	✓	✓	✓	✓
Securing Ad Hoc Networks [ZH99]	✓		✓		✓	✓	(✓)
The Quest for Security in Ad Hoc Networks [HBC01]	✓		✓		✓	✓	
Secure Pebblenets [BHBR01]	✓	✓		✓	✓	✓	✓
SPINS: Security Protocols for Sensor Networks [PSW+01]		✓	✓	✓	(✓)	✓	✓
A Key-Management Scheme for Distributed Sensor Networks [EG02]	✓	✓			✓	✓	
Random Key Predistribution Schemes for Sensor Networks: q-composite [CPS03]	✓	✓				✓	
Random Key Predistribution Schemes for Sensor Networks: Multipath Key Reinforcement [CPS03]	✓	✓			✓	✓	
Random Key Predistribution Schemes for Sensor Networks: Random-pairwise [CPS03]	✓	✓	✓		✓	✓	
Establishing Pairwise Keys for Secure Communications in Ad Hoc Networks: A Probabilistic Appr. [ZXSJ03]	✓	✓	✓		✓	✓	
A Key Management Scheme for Wireless Sensor Networks Using Deployment Knowledge [DDH+04]	✓	✓			✓	✓	
Establishing Pairwise Keys in Distributed Sensor Networks: Random Subset Assignment [LN03]	✓		✓		✓	✓	
Establishing Pairwise Keys in Distributed Sensor Networks: Grid-Based Predistribution [LN03]	✓		✓		✓	✓	✓
PIKE: Peer Intermediaries for key establishment in sensor networks [CP05]	✓	✓	✓		✓	✓	✓
Basic Key Distribution Scheme [WHC04]	✓	✓	✓	✓	✓		✓
Fault-Tolerant Key Distribution S. [WHCM04]	✓	✓	✓	✓	✓	✓	✓
Extended Key Distribution Scheme [WKHR05]	✓	✓	✓	✓	✓	✓	✓

Figure 7.3: Related work comparison

8

Conclusion and Outlook

In this chapter, we summarize the most important findings presented in this dissertation (Section 8.1) and give an outlook on interesting future work areas in Section 8.2.

8.1 Conclusion

As wireless sensor networks are used in more and more scenarios, there will always be new challenges coming toward us and demanding our attention. With this work we faced one important such challenge, namely providing secure key distribution for highly resource-constrained devices. Our key distribution schemes provide a foundation on top of which wireless sensor networks can be secured effectively and efficiently, making them suitable for a multitude of different application fields.

In this work, we presented three novel distributed key distribution schemes for wireless sensor networks based on symmetric cryptography. We provided a mathematical model for our schemes based on graph theory. This allowed us to adapt graph theory definitions and theorems, namely the concept of s-connectivity of a graph and the Menger theorem [Men27] to formally prove properties of the underlying key graph. All our presented schemes are based on the construction of an s-connected key graph, guaranteeing the existence of s node-disjoint paths in the graph. Using these node-disjoint paths, key shares are exchanged to construct new symmetric keys between arbitrary devices securely. By changing s, the security level of the network can be adapted as needed, mirroring different security requirements and risk levels.

Our key distribution schemes are fully inline with the requirements presented in Section 2.2. In the following we once again address each of these requirements and discuss briefly how our schemes fulfill it. The first requirement for a key distribution scheme for wireless sensor

networks is the need for decentralized operation. Our schemes work completely decentralized without the need for any central authority. Hence, by avoiding such a single point of failure our network remains operational even if some arbitrary devices fail or get subverted by an attacker.

The second requirement is the usage of symmetric cryptography only. Our algorithms operate without the need for any asymmetric cryptographic or polynomial-based operations, which would require high computational overhead on the devices. Instead they rely on symmetric cryptography to secure the transmission of key shares between neighboring nodes in the key graph. This makes them usable on highly resource-constrained devices as usually used for sensors and/or actuators, which need to be cheap and small. Furthermore, by using only symmetric cryptography, even these highly resource-constrained devices have enough resources left despite cryptography to perform their actual functionality.

Our key distribution schemes provide a graceful degradation of a network's security if an attacker subverts more and more devices, thus fulfilling our third requirement. As already mentioned, if some devices fail or get subverted by an attacker, the rest of the network stays fully operational. As long as the attacker cannot subvert at least s devices, new keys are guaranteed to be established securely. If the attacker subverts more and more devices, the network stays operational, while the probability of the secrecy of new keys decreases slowly (see Section 6.3.1), leading to the aforementioned graceful degradation of the network's security.

In contrast to many other approaches (e.g. [CP05, EG02]) our key distribution schemes do not rely on a fixed or predetermined network size. We provided algorithms for dynamically adding and removing devices at runtime of the network. Our algorithms can be tuned to use only constant memory on each device and only linear communication overhead. This is the foundation needed for scalability. Hence, our key distribution schemes operate with dynamic network sizes and scale for large networks, fulfilling our forth requirement.

Our fifth requirement is the ability to cope with device failures. We analyzed device failures and their effects on the key graph connectivity and the key establishment algorithm in detail. Devices in sensor and actuator networks may fail, e.g. due to depleted batteries or physical stress placed on the devices. We provided extensions to our algorithms making them immune against any kind of fail-stop device failure. In addition, our algorithms can cope with the concurrent failure of multiple devices, i.e. the rest of the network remains fully operational.

Finally, due to the underlying graph theory we can guarantee that whenever two devices of the network need to establish a new key, they are able to do so. Hence, our key distribution schemes provide a guarantee that any pair of nodes may establish a secure connection, fulfilling the sixth and last requirement.

8.2 Outlook and Future Research Activities

Even though our key distribution schemes provide a clear advancement over existing solutions, there are still some open research questions available that are worthwhile investigating. In the following we mention some of these possible future research areas.

In Section 2.1 we presented three different types of attackers namely Eve, Trudy and Mallory. While the first two attackers are completely covered and analyzed in this work, the analysis of Mallory is part of future work. An Mallory-type attacker is a very powerful attacker, since its goal is simply to hinder the normal network operation at all costs. In general coping with such a powerful attacker type is very hard or even impossible in asynchronous distributed systems.

The efficiency of our key distribution schemes is heavily dependent on the structure of the underlying key graph. As an example, one favorable structure for our extended approach is the balanced key graph as described in Section 5.6. While we provided rules for constructing such a key graph, it remains subject to future research to develop a distributed algorithm for the fully automated construction of a balanced key graph. In addition, in order to improve latencies and communication overhead it would be advantageous if the key graph is formed according to the underlying physical neighborhood graph. This could be achieved with key re-distribution: After deployment the devices would rebuild – inline with our construction algorithm – the key graph so that it matches the physical topology. Such key re-distribution mechanisms have not been in the focus of research so far and therefore almost completely new research territory.

Another interesting research domain would be joining and splitting key graphs. This is of particularly interest if e.g. a sensor and/or actuator network is used for home automation. In this case sensors deployed in the user's home could form one key graph domain while sensors deployed in the user's car could form the other. When the user arrives at home, both sensor networks can be merged. The question which arises is how to construct the key graphs in a way that if separated each works independently and if connected with each other secure communication is also possible between any two devices from any key graph.

While for fixed networks a number of intrusion detection systems have been proposed, this is still an area of research for wireless sensor networks. Our schemes are able to remove malicious nodes once they have been detected. It would be a challenging task to combine our results with an adequate intrusion detection system, which could actively detect and thereafter remove malicious nodes from the network.

Bibliography

[AES01] *Specification for the Advanced Encryption Standard (AES)*, volume 197 of *Federal Information Processing Standards publication*. U.S. National Bureau of Standards, Gaithersburg, MD, USA, 2001.

[AK96] Ross Anderson and Markus Kuhn. Tamper resistance - a cautionary note. In *Proceedings of the Second Usenix Workshop on Electronic Commerce*, pages 1–11, November 1996.

[BBB04] Jenna Burrell, Tim Brooke, and Richard Beckwith. Vineyard computing: Sensor networks in agricultural production. *IEEE Pervasive Computing*, 3(1):38–45, January-March 2004.

[Bha97] Ramesh Bhandari. Optimal physical diversity algorithms and survivable networks. In *ISCC '97: Proceedings of the 2nd IEEE Symposium on Computers and Communications (ISCC '97)*, page 433, Washington, DC, USA, 1997. IEEE Computer Society.

[BHBR01] Stefano Basagni, Kris Herrin, Danilo Bruschi, and Emilia Rosti. Secure pebblenets. In *Proceedings of the ACM Symposium on Mobile Ad Hoc Networking and Computing*, pages 156–163, 2001.

[Blo85] R Blom. An optimal class of symmetric key generation systems. In *Proc. of the EUROCRYPT 84 workshop on Advances in cryptology: theory and application of cryptographic techniques*, pages 335–338, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[BMJ+98] J. Broch, D.A. Maltz, D.B. Johnson, Y.C. Hu, and J. Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *In Proceedings of ACM/IEEE Mobicom*, pages 85–97, 1998.

[Bos95] Karl Bosch. *Elementare Einführung in die Wahrscheinlichkeitsrechnung*. vieweg studium, 6 edition, 1995.

[BSH+93] Carlo Blundo, Alfredo De Santis, Amir Herzberg, Shay Kutten, Ugo Vaccaro, and Moti Yung. Perfectly-secure key distribution for dynamic conferences. In *CRYPTO '92: Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, pages 471–486, London, UK, 1993. Springer-Verlag.

[Che77] R. V. Cherkasky. Algorithm of construction of maximal flow in networks with complexity of $O(V^2\sqrt{E})$ operations. *Mathematical Methods of Solution of Economical Problems*, 7:112–125, 1977.

[CKGLA90] C. Cheng, S.P.R. Kumar, and J.J. Garcia-Luna-Aceves. A distributed algorithm for finding k disjoint paths of minimum total length. In *28th Annual Allerton Conference on Communication, Control, and Computing*, Urbana, Illinois, October 1990.

[CKM00] D. Carman, P. Kruus, and B. Matt. Constraints and approaches for distributed sensor network security. Technical Report #00-010, NAI Labs, September 2000.

[CP05] Haowen Chan and Adrian Perrig. PIKE: Peer intermediaries for key establishment in sensor networks. In *Proceedings of IEEE Infocom*, March 2005.

[CPS03] Haowen Chan, Adrian Perrig, and Dawn Song. Random key predistribution schemes for sensor networks. In *IEEE Symposium on Security and Privacy*, May 2003.

[CWKS97] B. P. Crow, I. Widjaja, L. G. Kim, and P. T. Sakai. Ieee 802.11 wireless local area networks. *Communications Magazine, IEEE*, 35(9):116–126, 1997.

[DA99] T. Dierks and C. Allen. The tls protocol (version 1.0). IETF RFC 2246, January 1999.

[DDH+04] W. Du, J. Deng, Y. Han, S. Chen, and P. Varshney. A key management scheme for wireless sensor networks using deployment knowledge. In *INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*, page pp. 597. IEEE, March 2004.

[DES77] *Data Encryption Standard*, volume 46 of *Federal Information Processing Standards publication*. U.S. National Bureau of Standards, Gaithersburg, MD, USA, 1977.

[DF89] Yvo G. Desmedt and Yair Frankel. Threshold cryptosystems. In *CRYPTO '89: Proceedings on Advances in cryptology*, pages 307–315, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[Din70] E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Mathematics Doklady*, 11:1277–1280, 1970.

[DR02] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.

[DS05] David M. Doolina and Nicholas Sitar. Wireless sensors for wildfire monitoring. In *Proceedings of SPIE Symposium on Smart Structures & Materials (NDE 2005)*, Mar 2005.

[DvOW92] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, 1992.

[EG02] Laurent Eschenauer and Virgil D. Gligor. A key-management scheme for distributed sensor networks. In *Proceedings of the 9th ACM Conference on Computer and Communication Security (CCS-02)*, pages 41–47, November 18–22 2002.

[EK72] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.

[ET75] Shimon Even and R. Endre Tarjan. Network flow and testing graph connectivity. *SIAM Journal on Computing*, 4(4):507–518, December 1975.

[FF57] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Can. J. Math.*, 8:399–404, 1957.

[FKK96] A.O. Freier, P. Karlton, and P.C. Kocher. The ssl protocol version 3.0, March 1996.

[Gir05] Alexander Girgis. Implementation and Evaluation of Asymmetric Cryptographic Operations for the Smart-Its Platform. Study Thesis, Universität Stuttgart, 2005.

[GKvM03] Y. Guo, F. Kuipers, and P. van Mieghem. Link disjoint paths for reliable qos routing. In *International Journal of Communication Systems*, 2003. to appear.

[Gol98] Goldberg. Recent developments in maximum flow algorithms. In *SWAT: Scandinavian Workshop on Algorithm Theory*, 1998.

[Gon93] Li Gong. Increasing availability and security of an authentication service. *IEEE Journal on Selected Areas in Communications*, 11(5):657–662, 1993.

[GR98] Andrew V. Goldberg and Satish Rao. Beyond the flow decomposition barrier. *J. ACM*, 45(5):783–797, 1998.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.

[Har95] Frank Harary. *Graph Theory*. Perseus Publishing, 1995.

[HBC01] Jean-Pierre Hubaux, Levente Buttyan, and Srdan Capkun. The quest for security in mobile ad hoc networks. In *Proceeding of the ACM Symposium on Mobile Ad Hoc Networking and Computing (MobiHOC)*, pages 146–155, 2001.

[HC98] D. Harkins and D. Carrel. The internet key exchange (IKE). IETF RFC 2409, November 1998.

[HMTR04] Daniel Herrscher, Steffen Maier, Jing Tian, and Kurt Rothermel. A Novel Approach to Evaluating Implementations of Location-Based Software. In *Proceedings of the 2004 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2004)*, pages 484–490, San Jose, CA, USA, July 25–29 2004.

[IR84] A. Itah and M. Rodeh. The multi-tree approach to reliability in distributed networks. In *Proceedings of the 25th Symposium on FOCS*, 1984.

[Kar74] A.V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Mathematics Doklady*, 15:434–437, 1974.

[KN93] J. Kohl and C. Neuman. The Kerberos network authentication service (v5). IETF RFC 1510, September 1993.

[KS98] Stavros G. Kolliopoulos and Clifford Stein. Approximating disjoint-path algorithms using greedy algorithms and packing integer programs. In *IPCO*, 1998.

[KSW04] Chris Karlof, Naveen Sastry, and David Wagner. Tinysec: a link layer security architecture for wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 162–175, New York, NY, USA, November 3–5 2004. ACM Press.

[KW03] Chris Karlof and David Wagner. Secure routing in wireless sensor networks: Attacks and countermeasures. *Elsevier's AdHoc Networks Journal, Special Issue on Sensor Network Applications and Protocols*, 1(2–3):293–315, September 2003.

[LG01] S. J. Lee and Mario Gerla. Split multipath routing with maximally disjoint paths in ad hoc networks. In *IEEE International Conference on Communications*, pages 3201–3205, 2001.

[LLZ05] Guanfeng Li, Hui Ling, and Taieb Znati. Path key establishment using multiple secured paths in wireless sensor networks. In *CoNEXT'05: Proceedings of the 2005 ACM conference on Emerging network experiment and technology*, pages 43–49, New York, NY, USA, 2005. ACM Press.

[LN03] Donggang Liu and Peng Ning. Establishing pairwise keys in distributed sensor networks. In *10th ACM Conference on Computer and Communications Security (CCS '03)*, Washington D.C., October 2003.

[LNL05] Donggang Liu, Peng Ning, and Rongfang Li. Establishing pairwise keys in distributed sensor networks. *ACM Trans. Inf. Syst. Secur.*, 8(1):41–77, 2005.

[MCP+02] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, New York, NY, USA, 2002. ACM Press.

[MD01] Mahesh K. Marina and Samir R. Das. On-demand multipath distance vector routing in ad hoc networks. In *International Conference for Network Protocols*, 2001.

[Men27] K. Menger. Zur allgemeinen Kurventheorie. *Fund. Math.*, (10):96–115, 1927.

[Mil86] Victor S. Miller. Use of elliptic curves in cryptography. In *Lecture notes in computer sciences; 218 on Advances in cryptology—CRYPTO 85*, pages 417–426, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[Mot] Crossbow Technology Inc.: Motes, Smart Dust Sensors, Wireless Sensor Networks. Webpage. http://www.xbow.com.

[NET] NET - Network Emulation Testbed. Webpage. http://net.informatik.uni-stuttgart.de/.

[NS78] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.

[ORS93] R.G. Ogier, V. Rutenburg, and N. Shacham. Distributed algorithms for computing shortest pairs of disjoint paths. *IEEE Transactions on Information Theory*, 39(2):443–455, Mar 1993.

[OS89] R. Ogier and N. Shacham. A distributed algorithm for finding shortest pairs of disjoint paths. In *Proceedings of the Eighth Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE INFOCOM '89*, pages 173–182. IEEE, IEEE Press, Apr 1989.

[PK79] Gerald J. Popek and Charles S. Kline. Encryption and secure computer networks. *ACM Comput. Surv.*, 11(4):331–356, 1979.

[PSW+01] Adrian Perrig, Robert Szewczyk, Victor Wen, David E. Culler, and J. D. Tygar. SPINS: Security protocols for sensor networks. In *Mobile Computing and Networking*, pages 189–199, 2001.

[RLWW97] Heike Ripphausen-Lipa, Dorothea Wagner, and Karsten Weihe. The vertex-disjoint menger problem in planar graphs. *SIAM J. Comput.*, 26(2):331–349, 1997.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[SA99] Frank Stajano and Ross Anderson. The resurrecting duckling: Security issues for ad-hoc wireless networks. In *7th International Workshop on Security Protocols*, LNCS, pages 172–194, 1999.

[SAN90] Deepinder Sidhu, Shukri Abdallah, and Raj Nair. A distance vector algorithm for alternate path routing., 1990. submitted for publication.

[Sch95] Bruce Schneier. *Applied Cryptography*. J. Wiley and Sons, 2nd edition, 1995.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[SM03] Anand Srinivas and Eytan Modiano. Minimum energy disjoint path routing in wireless ad hoc networks. In *ACM Mobicom*, 2003.

[Sma] Home of the Smart-Its project. Webpage. http://smart-its.teco.edu.

[Smi03] Eric Smith. Eric's crypto software - DES for the microchip pic. Webpage, 2003. http://www.brouhaha.com/~eric/crypto/.

[SNA91] Deepinder Sidhu, Raj Nair, and Shukri Abdallah. Finding disjoint paths in networks. *SIGCOMM Comput. Commun. Rev.*, 21(4):43–51, 1991.

[Suu74] J. W. Suurballe. Disjoint paths in a network. *Networks*, 4:125–144, 1974.

[Wei91] Mark Weiser. The computer for the twenty-first century. *Scientific American*, 265(3):94–104, September 1991.

[WHC04] Arno Wacker, Timo Heiber, and Holger Cermann. A key-distribution scheme for wireless home automation networks. In *Proceedings of IEEE CCNC 2004*, Las Vegas, Nevada, USA, January, 5-8 2004. IEEE Communications Society, IEEE.

[WHCM04] Arno Wacker, Timo Heiber, Holger Cermann, and Pedro José Marrón. A fault-tolerant key-distribution scheme for securing wireless ad-hoc networks. In *Proceedings of the second Conference on Pervasive Computing, Pervasive 2004*, Vienna, Austria, April, 19-23 2004. Springer-Verlag.

[WKHR05] Arno Wacker, Mirko Knoll, Timo Heiber, and Kurt Rothermel. A new approach for establishing pairwise keys for securing wireless sensor networks. In *Proceedings of ACM SenSys'05*, San Diego, California, USA, November, 2–4 2005. ACM Press.

[Wur03] Ulrich Wurst. Einsatz kryptographischer Verfahren auf stark ressourcenbeschränkten Geräten. Diploma Thesis 2160, Universität Stuttgart, 2003.

[X.597] ITU-T Recommendation X.509: Information technology - open systems interconnection - the directory: Authentication framework, June 1997.

[XCX+04] Dahai Xu, Yang Chen, Yizhi Xiong, Chunming Qiao, and Xin He. On finding disjoint paths in single and dual link cost networks. In *IEEE Infocom*, 2004.

[ZH99] Lidong Zhou and Zygmunt J. Haas. Securing ad hoc networks. *IEEE Network*, 13(6):24–30, 1999.

[ZXSJ03] Sencun Zhu, Shouhuai Xu, Sanjeev Setia, and Sushil Jajodia. Establishing pairwise keys for secure communication in ad hoc networks: A probabilistic approach. Technical Report ISE-TR-03-01, George Mason University, March 2003.