INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Evaluation von Multicast Authentifizierung in Smart Homes mit dem TESLA Protokoll

Christopher Schütze

11. November 2018

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Evaluation von Multicast Authentifizierung in Smart Homes mit dem TESLA Protokoll

Christopher Schütze

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Dr. Nils gentschen Felde Tobias Guggenmos
Abgabetermin:	13. November 2018

I assure the single handed composition of this bachelor's thesis only supported by declared resources.

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 13. November 2018

. . . . . .
(Unterschrift des Kandidaten)

Abstract

Die Anzahl intelligenter Geräte wächst täglich und ist nicht mehr aus dem alltäglichen Leben wegzudenken. Gleichzeitig wachsen ebenfalls die Angriffsvektoren, weshalb umso wichtiger ist den Datenverkehr zwischen intelligenten Geräten zu schüzen. Dabei ist die Absicherung von Ende-zu-Ende Kommunikationen kein großes Problem, die Absicherung von Multicast-Datenverkehr im Gegensatz dazu eine fortwährende Herausforderungen. Eine Lösung zur Authentifizierung von Daten in einem Multicastszenario bietet das Protokoll TESLA. Diese Arbeit untersucht ob eine Verwendung von TESLA in einem speziellen Smart Home Szenario möglich ist. Dazu wird das Protokoll in das etablierte Transportprotokoll ESP aus der IPsec Familie eingebettet. Mit Hilfe einer im Rahmer dieser Arbeit entstandenen prototypischen Implementierung, wird TESLA in verschiedenen Anwendungsfällen, die einem Smart Home Szenario nachempfunden sind, getestet um eine Aussage darüber treffen zu können ob eine prinzipielle Verwendung möglich ist. Desweiteren wird, mit dem Fokus auf intelligente Geräte mit eingeschränkt verfügbaren Ressourcen, der Speicherbedarf des Prototypen analysiert. Abschließend kommt man zu der Aussage, dass das Protokoll TESLA eingebettet in ESP zur Absicherung von Multicast-Datenverkehr in einem Smart Home Szenario verwendet werden kann.

RSA Asymmetrisches kryptografisches Verfahren von **R**ivest, **S**hamir und **A**dleman

ECC Elliptic Curve Cryptography

DSA Digital Signature Algorithm

IPsec Security Architecture for the Internet Protocol

AH Authentication Header

ESP Encapsulating Security Payload

ICV Integrity Check Value

SA Security Association

SPI Security Parameters Index

SAD Security Association Database

IKE Internet Key Exchange

IoT Internet of Things

Riot-OS Betriebsystem für Internet of Things

IPv4 Internet-Protokoll Version 4

TTL Time-To-Live

IPv6 Internet-Protokoll Version 6

TESLA Timed Efficient Stream Loss-Tolerant Authentication

PSK Pre-Shared-Key

RAM Arbeitsspeicher

ROM Flashspeicher

CPU Prozessor

RFC Request for Comments

CoAP Constrained Application Protokoll

UDP User Datagram Protocol

GCC Gnu Compiler Collection

MAC **M**essage **A**uthentication **C**ode

RTT Round-Trip-Time

PRF Pseudo Random Function

HMAC Keyed-Hash Message Authentication Code [19]

GNRC Generic Network Stack

CBC Cipher Block Chaining

ICMP Internet Control Message Protocol

KB Kilobyte (1 KB = 1024 Byte)

MHz Mega-Herz

EMSS Efficient Multi-chained Stream Signature

inf**TESLA** $infinity$-TESLA

DoS Denial-of-Service

PKI Public Key Infrastructure

MS Millisekunde(n)

G-IKEv2 Group IKEv2

Inhaltsverzeichnis

1 Einleitung

Moderne Computer sind aus dem heutigen Alltag nicht mehr weg zu denken und täglich wächst die Zahl an Geräten welche sich vernetzen und über das Netzwerk kommunizieren. Dazu zählen neben modernen Computern im klassischen Sinn, wie zum Beispiel Desktop-PCs, Latops oder leistungsstarke Server in Rechenzentren auch intelligente Geräte (*Smart Devices*) wie mobile Telefone.

Smart Devices sind in der Regel alle Geräte, welche in ihrer grundlegenden Funktion keine Intelligenz besitzen, diese jedoch durch zusätzliche Soft- und Hardware verliehen bekommen. Wird zum Beispiel eine Uhr um Bewegungs- und Vitalsensor sowie um eine Kommunikationschnittstelle ergänzt, bietet sie bereits weitere Funktionalitäten als ursprünglich. Um die Sensordaten auszulesen, zu interpretieren und über die Kommunikationsschnittstelle an andere Geräte zu senden, muss zusätzlich Software bereitgestellt werden. Durch die Erweiterung von Hard- und Software sowie der Möglichkeit die neuen Informationen, welche aus der zusätzlichen Hardware hervorgehen, zu verarbeiten, wird die Uhr zunehmenden intelligenter.

Neben sogenannten *Wearables* werden auch größeren Objekten, wie Privathäusern und Intrustrieanlagen, nahezu täglich intelligenter. Zum Beispiel ist es in einem modernen Haus möglich, verschiedene Geräte über ein Tablet oder ein mobiles Telefon anzusprechen und zu steuern. In Industriesteueranlagen finden sich offentlichtlich sicherheitskritische Anwendungensfälle wie Sensoren an Geräten oder komplexe Feuerlöschsysteme, welche nicht nur digital verwaltet und überwacht sondern ebenfalls gesteuert werden müssen. Grundlegend unterscheiden sich solch eingebettete Systeme nicht zu herkömlichen Computersystemen, verfügen aber bedingt durch den Verbauort und geringere Größe des Systems nur über eingeschränkte Ressourcenkapazitäten. Zudem sind eingebettete Systeme nur eingeschränkt mit Energie versorgt, da in Abhänigkeit zum Verbautort unter Umständen keine konstante und unbegrentze Energieversorgung gewährleistet werden kann.
Unabhängig vom Verwendungszweck und Anwendungsfall werden an eingebettete Systeme bestimmte Mindestanforderungen bezüglich der Datenübertragung gestellt, welche sich nicht zu etablierten Sicherheitsanforderungen für herkömlichen Computersystemen unterscheidet.

Authentizität [9] beschreibt in der Informationsicherheit die Echtheit und die Vertrauenswürdigkeit von Daten, welche mittels Überprüfung nachgewiesen werden kann. Bei der Überprüfung auf Authentizität kann der Datenursprung nachgewiesen werden, wodurch Daten einem angegebenen Sender nicht abstreitbar zugeordnet werden können.

Integrität [9] beschreibt die Korrektheit von Daten und lässt eine Aussage darüber zu ob Daten korrekt und unverändert empfangenen worden sind. Da die Aussagekraft von Integrität alleinstehend nicht sinnvoll ist sollte immer das Ergebnis der Authentizzätsprüfung miteinbezogen werden, um final feststellen zu können, ob die zu prüfenden Daten weiter verarbeitet werden sollten.

Um sich gegen mögliche Angriffe zu schützen wurden verschiedene Mechanismen und Algorithmen entwickelt, welche zum Einen das Mitlesen verhindern können oder zum Anderen das Erkennen von etwaigen Manipulation der transferierten Daten ermöglichen. Zwei etablierte und moderne kryptografische Verfahren sind DSA und RSA [24], die das mathematischen Problem von großen Primzahlen zu Grunde legen um eine möglichst hohe Sicherheit zu gewährleisten. Durch stetigen Fortschritt der Technik bei Angriffen ist es allerdings notwendig immer größere Schlüsselängen zu verwenden, wodurch die bereits rechen- und speicherintensiven Verfahren noch mehr Ressourcen beanspruchen. Eine Alternativ hierzu bietet Elliptic Curve Cryptography (ECC) [22], die bei geringeren Schlüssellängen und daraus folgendem geringeren Rechen- und Speicherbedarf, ein äquivalentes Sicherheitsniveau erreicht. Dabei gibt das Sicherheitsniveau an, wie hoch der Aufwand zum Lösen des Problems ist, ohne die benötigten Parameter zu kennen. Dies zeigt *Scott A. Vanstone* in seinem Paper *Next generation security for wireless: elliptic curve cryptography - Tabelle 3* [33] in dem verschiedenen kryptografische Verfahren gegenübergestellt und anhand einer festen Anzahl von Sicherheitsbits verglichen werden. Daraus geht hervor, dass bereits bei 128 Bits DSA und RSA signifikant längere Schlüssel benötigen und ECC deutlich effizienter und ressourcenoptimierter arbeitet. Um das gleiche Sicherheitsniveau zu erreichen, verlangen DSA und RSA mindestens 3072 Bits wohingegen ECC bereits mit 256 Bits auskommt.
Alleinstehend sind kryptografische Verfahren allerdings für die Absicherung von Kommunkationskanälen wenig nützlich, da sie keine Vorgehen zur Datenübertragung spezifizieren. Weshalb sie ihre Anwendung in Übertragunsprotokollen finden, welche dem Schutz des IP-Netzwerkverkehrs dienen. Wobei für das IPv4-Protokoll keine Protokolle auf OSI-Layer 3 existieren, sind für das IPv6-Protokoll gleich zwei Protokoll in der Protokoll-Familie IPsec [18] spezifiziert, welche im Kapitel 2.1 detailierter erklärt werden.

Authentication Header (AH) [16] stellt die Authentizität und Integrität des übertragenen Datenpakets sicher, indem eine Prüfsumme über alle Invarianten des IP-Pakets gebildet wird. Einen Schutz vor unauthorisiertem Mitlesen bietet AH jedoch nicht, da die Nutzdaten nicht verschlüsselt werden.

Encapsulating Security Payload (ESP) [17] stellt neben Authentizität und Integrität auch die Vertraulichkeit [9] sicher um ein unauthorisiertes Mitlesen Dritter verhindern zu können. Anders als beim AH werden die Invarianten Felder des IP-Pakets nicht durch eine Prüfsumme geschützt, es besteht jedoch die Möglichkeit die Nutzdaten zu verschlüsseln.

Mit zunehmenden Einsatz von IPv6 werden auch die Protokolle von IPsec verwendet um die Sicherheit der zu übertragenen Daten im Rahmen einer Ende-zu-Ende-Kommunikation zu garantieren. Außerdem kann man behaupten, dass dies auf herkömlichen sowie eingebetteten Systemen zu realisieren ist sofern die Mindestanforderungen der nötigen Ressourcen zur Verwendung von Transportsicherheitsprotokollen erfüllt sind. Greift man nochmals das bereits erwähnte Szenario Smart Home auf, stellt sich zunächst die Frage ob die Absicherung der Kommunikation ebenso trivial ist. In einem solche Umfeld existiert eine Vielzahl von eingebetteten System, die verwaltet und gesteuert werden müssen. Angefangen von Außenrollos, über Heizungsanlagen und Haushaltsgeräten, diversen Lampen und anderen *Smart Devices* im Haushalt, bis hin zu verschiedensten Sensoren. All diese Systeme verfügen offentsichtlich über zum Teil stark eingeschränkte Ressourcenkapazitäten, was den Einsatz von bewährten Authentifizierungsmethoden deutlich erschwert. Weiter möchte man die Geräte nicht im-

mer einzeln ansprechen müssen, sondern mittels einem Datenpaket möglichst alle relevanten Geräte einreichen können.
Schickt man ein Datenpaket an eine Gruppe ausgewählter Empfänger spricht man von Multicasting. Dadurch ist es möglich in größeren Netzwerken tausende oder gar millionen Empfänger zu erreichen. Gleiches gilt auch für Angreifer, welshalb es umso wichtiger ist Multicast-Netzwerkverkehr zu schützen. Die Absicherung des Datenverkehrs in einem Multicast-Szenario stellt jedoch bislang eine Herausforderungen dar, weshalb sich die zuvor gestellten Frage, ob die Absicherung in einen Smart Home Szenario ebenso trivial ist, mit nein beantwortet lässt. Die Lösung des Problems ist zwar nicht trivial, es existieren allerdings Lösungsansätze die einen Mechanismus spezifizieren wie der Schutz bei der Datenübertragung mit Multicast realisiert werden kann. Eines dieser Verfahren ist das TESLA Protokoll [29]. Zum Erfüllen der Mindestanforderungen für eine sichere Verbindung zwischen Netzwerkentitäten, verwenden etablierte Verfahren asychrones Schlüsselmaterial, wobei Sender und Empfänger je einen öffentlichen und geheimen Schlüssel besitzen um sich zu authentifizieren und die zu sendenen Daten zu schützen. Obwohl Algorithmen wie ECC deutlich ressourcenoptimierter arbeiten, sind diese weiterhin sehr rechen- und speicherintensiv. Dadurch ist der Einsatz auf Geräten mit eingeschränkten Ressourcen inadäquat. TESLA löst dies in dem Prüfsummen über die Daten gebildet werden und verringert dadurch den benötigten Rechenaufwand und Speicherbedarf, da das Berechnen von Prüfsummen, unabhängig des gewählten Algorithmus, signifikant bessere Laufzeiten aufweist. Um dennoch die Sicherheitscharakteristik Asychronität zu bewahren, bildet TESLA diese auf die Zeit ab. Die Berechnung der kryptografischen Prüfsummen kann somit mit sychronem Schlüsselmaterial erfolgen, welche anschließend mit einem vom Anwender festgelegten zeitlichen Versatz an die Empfänger publiziert wird. Folglich stellt das Protokoll TESLA eine mögliche Problemlösung zur Authentifizierung von Multicast-Datenverkehr bereit.
Da in einem Smart Home jedoch unterschiedlichste Geräte zum Einsatz kommen, welche miteinander kommunizieren müssen stellt sich die Frage, ob das Protokoll TESLA praktisch einsetzbar ist. Betrachtet man den speziellen Anwendungsfall *Lichtschalter* in einem Smart Home Szenario näher, ergeben sich verschiedene Problemstellungen welche zu klären sind. Da ein Lichtschalter, welche als Sender aggiert, ein System einbetten kann welches deutlich größer sein darf als das, welches von Lampen eingebettet werden muss, gilt zu klären ob die zu verwendenden eingebetteten Systeme ausreichend Ressourcen zur Verfügung stellen um TESLA verwenden zu können. Weiter sind Lampen in einem Haushalt nicht immer im identischen Abstand zum zugehörigen Lichtschalter installiert. Wodurch weitergehend untersucht werden muss ob das Protokoll TESLA in einem Szenario in dem Empfänger (Lampen) in variable Abständen zum Sender (Lichtschalter) positioniert sind, vollumfänglich funktioniert.
Ziel dieser Arbeit ist es das Protokoll TESLA prototypisch für eingebettete System zu implementieren und zu untersuchen ob es in einem Smart Home Szenario verwendet werden kann um eine sichere Kommunikation zu gewährleisten. Außerdem soll geklärt werden mit welchen Arten von intelligenten Geräte TESLA kompatibel ist, wodurch sich abschließend beurteilen lässt ob ein Umsetzung für intelligente Lampen umsetzbar wäre.

2 Grundlagen

Im folgenden Kapitel werden zunächst die Grundlagen geklärt, welche für das Verständnis dieser Arbeit notwendig sind. Security Architecture for the Internet Protocol (IPsec) ist ein wichtiger Bestandteil dieser Arbeit sowie der prototypischen Umsetzung, weshalb zunächst die Protokoll-Familie IPsec detailierter betrachtet wird. Dabei werden die beinhalteten Protokolle, Mechanismen und Strukturen geklärt. Zudem wird aufgezeigt wie sich die Protokolle in Abhängigkeit der IPsec-Modi verschiedenen verhalten und welche Auswirkungen diese auf ein IP-Datagram haben. Anschließend wird aufgezeigt wie sich eingebette Systeme in verschiedene Klasse einordnen lassen und sich diese abgrenzen lassen. Dadurch ist es möglich zu bestimmen wie sich ein System klassifizieren lässt um eine Aussage darüber treffen zu können welcher Art System eine Anwendung für eingebettete Systeme mindestens benötigt um lauffähig zu sein. Dafür werden die einzelenen Klasse detailierter erläutert um neben der Ressourcehneinschränkungen die spezifischen Unterschiede aufzuzeigen. Weshalb die Wahl des Betriebsystems auf Riot-OS gefallen ist und somit die Basis der in dieser Arbeit entstanden Umsetzung darstellt, soll abschließend aufgezeigt werden. Hierfür wir Riot-OS einer Auswahl alternativen Betriebsystemen gegenübergestellt und kurz auf einscheidungsrelevante Kriterien eingegangen.

2.1 IP Security

IPsec [18] ist im OSI-Layer-Model auf Layer 3 angesiedelt und bietet unabhängig der darüber liegenden Schichten eine Schnittstelle zwischen unsicheren und sicheren Kanälen, um den IP-Netzwerkverkehr zu schützen. IPsec selbst bezeichnet eine Sammlung von Protokollen und Mechanismen, weshalb es als Protokoll-Familie zu betrachten ist. Eine Kernfunktionalität stellt dabei die Security Association (SA) dar und dient grundlegend der Verwaltung einer Verbindung zwischen Netzwerkentitäten sowie den ausgehandelten Sicherheitsparametern. Darauf aufbauend ist in IPsec das Schlüsselmanagmentprotokoll Internet Key Exchange (IKE) spezifiziert, welches zum Austausch von Schlüsselmaterial über unsichere Kanäle konzipiert ist. Zur Absicherung der Nutzdaten stellt IPsec die beiden Sicherheitsprotokolle AH und ESP bereit, welche in den Unterkapiteln 2.1.2 und 2.1.3 detailierter erläutert werden.
Zusätzlich kann bei der Verwendung von IPsec zwischen zwei Modi entschieden werden die unterschiedliche Einflüsse auf das IP-Datagramm haben.

Transport-Modus Verwendung findet dieser Modus überwiegend für Ende-zu-Ende Kommunikation und modifiziert das original IP-Datagram nur geringfügig, indem das Next-Header-Feld im IP-Header angepasst wird, welches den Typen des nachfolgenden Headers beschreibt. Dies ist notwendig, da der IPsec-Header zwischen IP-Header und Nutzdaten eingefügt wird.

Tunnel-Modus Dieser Modus findet seine Verwendung um zwei IP-Netze über einen unsicheren Kanal miteinander zu verbinden. Das original IP-Paket wird beibehalten und

in einem Neuem gekapselt, wodurch der IP-Header unverändert bleibt und das ursprüngliche IP-Paket zu den Nutzdaten wird.

2.1.1 Security Association

Die SA ist eine Struktur welche Sicherheitsattribute beinhaltet die beschreiben wie ein Host mit einem Anderem sicher Daten austauschen kann. Bedingt durch die Einschränkung, dass jeder SA immer nur einen Dienst und Verbindung beschreibt, ist sie eine simplex Vereinbarung zu einer Verbindung zwischen zwei Netzwerkentitäten. Eine eindeutige Beziehung zwischen einer Verbindung und der SA wird über den Security Parameters Index (SPI) hergestellt, welcher im IPsec Header an erster Stelle steht. Ist das nicht außenreichend präzise, wird zusätzlich über die Parameter Ziel- und Quelladresse referenziert, womit ein unikaler Eintrag in der Security Association Database (SAD) identifiziert werden. Ist kein Eintrag in der SAD zu finden, wird das eingehende Paket verworfen und nicht weiter verarbeitet. Neben der SPI, Ziel- und Quelladresse sind weitere Attribute, wie das Protokoll (Dienst), ausgehandelte kryptografische Algorithmen sowie die zugehörigen kryptografischen Schlüssel in der SA hinterlegt.

2.1.2 Authentication Header

AH [16] garantiert Authentizität und Integrität der übertragenen Daten und bietet mittels Sequenznummern einen Schutz vor Replay-Attacken. Die Sicherheit wird durch eine berechnete Prüfsumme gewährleistet, welche die Nutzdaten sowie alle Invarianten Felder des IP-Headers beinhaltet. Die Felder des IP-Headers, welche bei der Übertragung von Routern verändert werden können, fließen nicht mit in die Berechnung der Prüfsumme ein. In einem IPv6-Header bedeutet das den Auschluss des Feldes Hop-Limit, welches aus dem IPv4-Header als Time-To-Live (TTL) bekannt ist. Wie sich AH in Abhängigkeit des verwendeten Modus einordnet zeigt die Abbildung 2.1. Auf Grund der fehlenden Möglichkeit die Nutzdaten zu verschlüsseln bietet AH in der Regel nicht ausreichend Schutz und Flexibilität, woraus sich die seltene Verwendung ergibt wenngleich eine Kombination mit ESP technisch möglich ist.

2.1.3 Encapsulating Security Payload

Analog zu AH offeriert ESP [17] Authentizität und Integrität sowie einen Schutz vor Replay-Attacken, bietet jedoch zusätzlich die Möglichkeit der Verschlüsselung so dass ein unautho-risiertes Lesen verhindert werden kann. Dabei ist ESP flexibel im Einsatz der Sicherheitsmaßnahmen, in dem nur die Berechnung eines Integrity Check Value (ICV), nur die Datenverschlüsselung oder Beides angewendet werden kann. Der ICV umfasst ausschließlich die Felder des ESP-Headers und keine Felder des IP-Headers, welche durch die Kombination mit AH zusätzlich geschützt werden können. Somit kann ESP in verschieden Kombination mit und ohne AH verwendet werden.

- ICV + Verschlüsselung
- AH + ICV
- AH + Verschlüsselung
- AH + ICV + Verschlüsselung

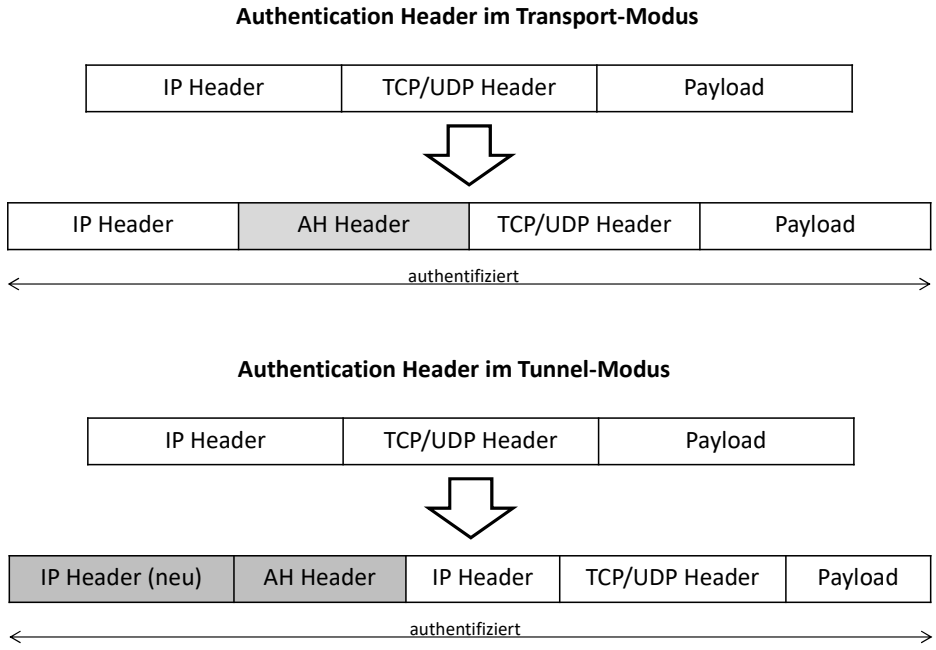

Abbildung 2.1: IPSec-Modus Authentication Header

Wie auch AH kann ESP in zwei verschiedenen Modi verwendet werden. Welche Auswirkungen dies auf das originale IP-Datagram hat und welche Teile des IP-Paketes authentifiziert und verschlüsselt sind zeigt die Abbildung 2.2.

2.1.4 Internet Key Exchange

IKE [15] ist für die Schlüsselverwaltung und den Schlüsselaustausch über unsichere Kanäle konzipiert. Vor der Verwendung einer abgesicherten Verbindung müssen Sicherheitsparameter, wie der zu verwendende kryptografische Algorithmus, vereinbart und gemeinsame Schlüssel ausgetauscht werden. In dieser Phase des Verbindungsaufbau authentisieren sich beide Entitäten gegeneinander. Die Authentesierung erfolgt entweder auf Basis eines zuvor verteilten Schlüssels Pre-Shared-Key (PSK) oder unter zur Hilfenahme von Zertifikate. Final werden alle ausgehandelten Parameter in einer SA abgelegt und anschließend für AH oder ESP verwendet.

2.2 Klassifizierung eingebetteter Systeme

Geschuldet der stetig wachsenden Anzahl von intelligente Geräten welche über das Internet verbunden sind, wird es fortwährend wichtiger Geräte verschiedener Klassen einzuordnen zu können. Ein mögliche Einordnung anhand den Charakteristika Arbeitsspeicher (RAM), Flashspeicher (ROM) und der Prozessor (CPU) wurde im Request for Comments (RFC) *Terminology for Constrained-Node Networks* [3] genau spezifiziert. Die Tabelle 2.1 ist der im RFC beschriebenen Klassifizierung nachempfunden und wird nachfolgend detailierter erläutert.

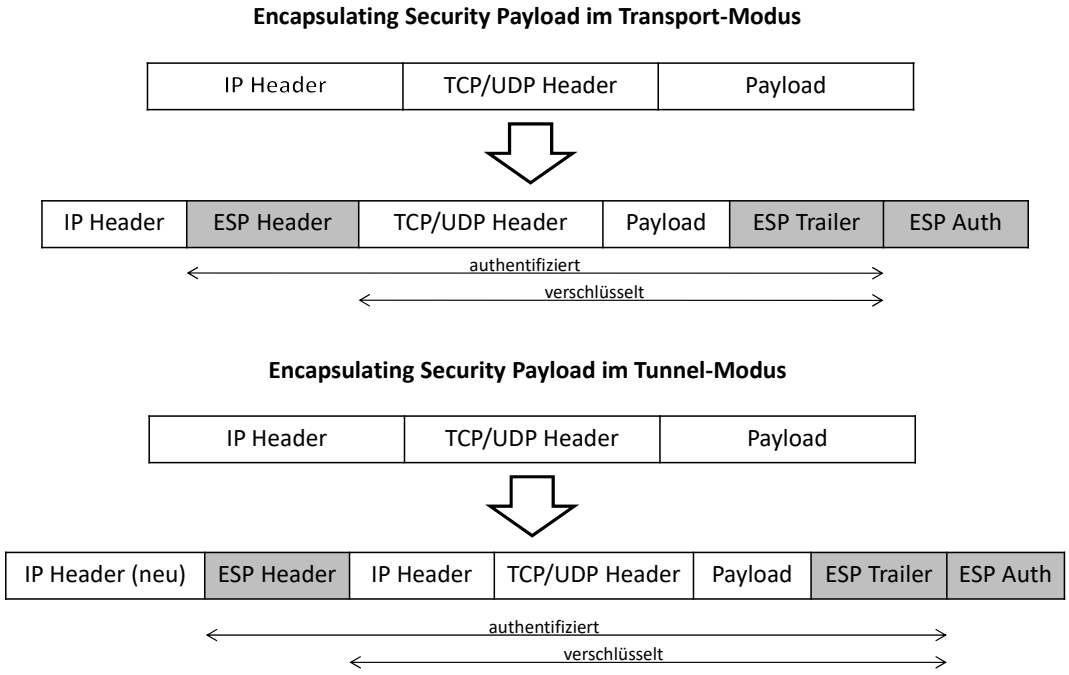

Abbildung 2.2: IPSec-Modus Encapsulating Security Payload

Name	**RAM (data size)**	**ROM (Code size)**
Class 0, C0	$<$ 10 KB	$<$ 100 KB
Class 1, C1	$\sim$ 10 KB	$\sim$ 100 KB
Class 2, C2	$\sim$ 50 KB	$\sim$ 250 KB

Tabelle 2.1: Classes of Contrained Devices (1 KB == 1024 Bytes)

Class 0 Geräte verfügen nur über extrem eingeschränkten Ressourcenkapazitäten und sind in der Mehrheit Sensoren ähnlich oder gar reine Sensoren. Auf Grund des stark eingeschränkten Speichers und der geringen Rechenleistung ist es diesen in der Regel nicht möglich gemäßt den Sicherheitsmindestanforderungen (Authentizität und Integrität) über das Internet zu kommunizieren. Sie interagieren mit stärkeren Gateway-Knoten um den Datentransfer via Internet zu realisieren. Generell sind diese Geräte mit einem sehr kleinen Datensatz vorkonfiguriert und stellen ausschließlich rudimentäre Funktionen (Einschalten, Ausschalten oder Gerätestatus) zur Verfügung.

Class 1 Geräte unterliegen etwas weniger starken Restriktionen als Class 0 Geräte, sind allerdings nicht auf Gateway-Knoten angewiesen welche die Internetkommunikation übernehmen müssen. Diese Klasse von Geräte sind potenziell in der Lage speziell für eingebettete Systeme entwickelte Protokolle wie Constrained Application Protokoll (CoAP) [31] über User Datagram Protocol (UDP) zu nutzen und somit in ein Netzwerk integrierbar.

Class 2 Geräte unterstützen grundlegend einen Großteil der etablierten Protokolle zur Kommunikation mit anderen Netzwerkentitäten, wodurch sie flexibler in bestehende Netze einzubinden sind. Somit stellen diese Geräte annähernd vollwertige Kommunikationsendpunkte dar, profitieren jedoch zugleich von einem leichtgewichtigen und Energieeffizienten Protokollstack wodurch weniger Speicher für den Datenaustauschen erforderlich ist und mehr Kapazitäten für Applikationen zur Verfügung stehen.

Im weiteren Verlauf werden diese Unterscheidungen als Klasse 0, Klasse 1 oder Klasse 2 bezeichnet. Selbstverständlich existieren Geräte, welches sich nicht in einer der oben aufgelisteten Klassen einordnen lassen, jedoch zum Beispiel auf Grund einer nicht vorhandene konstante Energieversorgung, als eingebettet System mit eingeschränkten Ressourcenkapazitäten zu bezeichnen sind. Diese Geräte unterstützen nahezu alle etablierten Netzwerkprotokolle ohne Einschränkungen, weshalb sie nicht spezieller betrachtet werden.

2.3 RIOT OS

Durch eine hohe Anzahl von Varietäten typischer Klasse 0 - 2 Geräten (siehe 2.2), sind die Bedingungen deutlich erschwert die zwei primären Aufgaben eines Betriebssystems, die Hardwareschnittenstellen zu abstrahieren und die Systemressourcen zu verwaltung, zu realisieren. Die Hardwarearchitekturen eingebetteter Systeme sind sehr heterogen und reichen von 8-Bit AVR [14] Microchip bis hin zu 32-Bit Cortex-M7 [14] Architekturen, weshalb viele IoT-Betriebssysteme teilweise nur eine dieser Architekturen unterstützen. Betriebsystem für Internet of Things (Riot-OS) [2] unterstützt dank einem hohem Abstraktiongrad der Hardwareschnittstellen eine Reihe von verschiedenen Architekturen. Wie im Abschnitt 2.2 aufgeführt, verfügen Klasse 0 und 1 Geräten über sehr wenig RAM und ROM. Die gegebenen Speichermaxima dürfen von einen Betriebsystem nicht überschritten werden und es müssen zugleich ausreichend Speicherreserven für Netzwerkstack und Applikationen zur Verfügung stehen. So ist, wie in Tabelle 2.2 abzulesen, Linux nicht für Klasse 0 - 2 Geräte geeignet. Der Kernel von Riot-OS ist voll modular aufgebaut wodurch weitesgehend Modulabhänigkeiten eliminiert werden und das System auf seinen speziellen Einsatz zugeschnitten werden kann. Folglich minimiert diese Eigenschaft den Speicherverbrauch des Betriesystems und wird den strengen Speicheranforderungen eniger Klasse 0 Geräte gerecht. Neben den bereits auf-

OS	Min RAM	Min ROM	C	C++	Multi-Threading	Modularity	Real-Time
Contiki	$<$ 2KB	$<$ 30KB	•	×	•	•	•
Tiny OS	$<$ 1KB	$<$ 4KB	×	×	•	×	×
Linux	$\sim$ 1MB	$\sim$ 1MB	✓	✓	✓	•	•
Riot-OS	$\sim$ 1.5KB	$\sim$ 5KB	✓	✓	✓	✓	✓

Tabelle 2.2: Betriebssystem Charakteristika. voll unterstützt(✓), teilweise unterstützt(•), nicht unterstützt(×), 1 MB = 1024 KB, 1 KB = 1024 Byte
Quelle: https://riot-os.org/ (Abgerufen: 8.9.2018)

geführten Anforderungen an Speichereffizienz und Hardwareabstraktion, werden weitergehend die Aspekte der verwendbaren Programmiersprache und Energieeffizienz betrachtet. Tabelle 2.2 zeigt, dass jediglich Linux und Riot-OS die Programmiersprachen C und C++ vollumfänglich unterstützen. Die unterstützung von C oder C++ ermöglicht das Einbin-

den und Verwenden externe Bibliotheken sowie das Kompilieren des Quellcodes mit dem C-Kompiler *Gnu Compiler Collection (GCC)* was die Benutzerfreundlichkeit für Entwickler deutlich erhöht. Die gegenübergestellten Betriebssysteme Contiki [8] und Tiny OS [20] hingegen, unterstützen nur eine Teilmenge von C-Schlüsselwörtern bzw nur den eigenen C-Dialekt (nesC [12]). Da eingebette Systeme in der Regel eine eingeschränkte Energieversorgung haben ist es wichtig, dass auch das Betriebssystem dahingehend optimiert den Energiebedarf gering zu halten. Die effizienteste Vorgehensweise hierbei ist, das gesamte System oder Teile davon, wie z.B. externe Module, in einen Ruhemodus zu versetzen oder komplett auszuschalten. Contiki, TinyOS und Riot-OS sind darauf ausgelegt energiesparend zu operieren setzen dies aber auf unterschiedliche Arten um. Contiki bieten eine Programmierschnittstelle, womit der Energieverbrauch kumuliert und gemessen werden kann. TinyOS definiert dafür Programmierkonvetionen, die es erlauben Subsysteme in einen Energiesparmodus zu setzen oder ein und aus zu schalten. Riot-OS implementiert nicht-zyklische Systemticks wodurch das System und angeschlossene Geräte beliebig lang in einen Ruhemodus gehalten werden können und nur mittels externen Interrupt verlassen werden. Hierbei bietet Riot-OS im Gegensatz zu Contiki oder TinyOS den Vorteil, dass sich der Entwickler nicht explizit um das Energiemanagment sorgen muss und es keine zusätzlichen Implementierungen benötigt.

3 Timed Efficient Stream Loss-Tolerant Authentication

Im Folgenden wird das Protokoll TESLA [29] näher erläutert. Entwickelt wurde es von Adriag Perrig, Ran Canetti, J.D. Tygar und Dawn Song und 2005 in dem Paper *The TESLA broadcast authentication protocol* [28] veröffentlich. Darin stellen die Authoren folgende fünf Anforderung an ein Protokoll zur Authentifizierung von Broadcastpaketen, welche TESLA als solches vollumfänglich erfüllt.

1. niedriger Rechenaufwand für die Authentifizierung
2. niedriger Kommunikationaufwand zwischen Netzwerkentitäten
3. limitiertes Puffern auf Sender- und Empfängerseite
4. tollerant gegenüber Paketverlust
5. skalierbar für hohe Empfängeranzahl

Die grundlegende Idee von TESLA ist, dass der Sender für die zu übertragenen Daten eine Prüfsumme (auch **M**essage **A**uthentication **C**ode (MAC)) mittels einem arkanen und nur für den Sender bekannten Schlüssel k berechnet und diese mit den Daten konkateniert. Der Empfänger puffert die eingehenden Pakete und hält sie vor, bis der Sender den Schlüssel k nach einer bestimmten Zeit veröffentlicht. Ist der Schlüssel k beim Empfänger bekannt, kann dieser die vorgehaltenen Pakete authentifizieren. Ein simples Vorgehen wäre demnach jedem Paket einen eigenen Schlüssel zu zuweisen. TESLA definiert weitere spezifische Anforderung um selbst flexibler zu aggieren. Somit müssen Sender und Empfänger lose zeitsychronisiert sein, was die Verwendung eines Zeitsychronisationsprotokoll (z.B. NTP [23]) weder erfordert noch erzwingt. Weiter muss beim Empfang von einem Paket zunächst der angehängte Schlüssel k verifiziert werden. Dies kann mit eventuell bereits erhaltenen Schlüssel geschehen. Ist der Schlüssel k korrekt so ist es anschließend dem Empfänger möglich, die MACs der zwischengespeicherten Pakete zu verifizieren. Bedingt durch die einfache Zeitsychronisation kann der Empfänger außerdem feststellen ob der Schlüssel k für ein Paket bereits veröffentlicht wurde. Aus in der Initialisierungsphase vom Sender mitgeteilten Protokollkonfiguration, kann ausgelesen werden mit welchem Zeitversatz die Schlüssel veröffentlicht werden, wodurch der Empfänger in der Lage ist zu berechnen ob für ein Paket der Schlüssel k bereits bekannt sein kann oder nicht. Dieser Zeitversatz definiert der Sender zum Startzeitpunkt fest. Voraussetzung hierfür ist, dass beim Initialisieren des Senders eine Zeitspanne definiert die die Länge eines Zeitintervals beschreibt. Diese Information liegt nach der Initialierungsphase beim Empfänger vor womit dieser nahezu eindeutig bestimmten kann wie groß der zeitliche Abstand zwischen zwei Schlüsselveröffentlichungen ist. Der Versatz wird in Intervallen angegeben und wird mit der Zeit des Interval multipliziert werden um die zeitliche Verzögerung zu bestimmen.

Für die von TESLA speziell geforderte lose Zeitsychronisation ist eine einfache Zwei-Wege-Sychronisation spezifiziert, welches zunächst unter 3.1 erklärt wird. Zur Berechnung des MAC wird, wie bereits beschrieben ein Schlüssel k benötigt. Dieser Schlüssel ist Teil einer Kette von Schüsseln, wobei das Konstrukt der verketteten Schlüssel als *One-Way-Key Chain* bezeichet und unter 3.2 näher erläutert wird. Nachdem die Kernfunktionalitäten für TESLA geklärt sind, werden im Anschluss die Initialisierungphasen für Sender 3.3 und Empfänger 3.4 beschrieben. Ohne dass diese beiden Entitäten vollständig initialisiert sind, ist es ihnen nicht möglich mittels TESLA untereinander zu kommunizieren. Das Ziel von TESLA ist die authentifizierte Kommunikation, weshalb abschließend die Vorgehensweise auf Sender- und Empfängerseite unter 3.5 geklärt wird.
Aus der Spezifikation von TESLA geht außerdem hervor, dass die Initialisierungphase über einen sicheren Kanal abgehandelt werden muss. Wie dieser aussehen muss, wird nicht weiter erläutert und obliegt dem Benutzer, weshalb dies nicht in diesem Kapitel aufgegriffen wird.

3.1 Zeitsychronisierung

Im Folgenden wird das Vorgehen der Zeitsychronisation, welches TESLA spezifiziert näher erläutert. Explizit ist keine starke Sychronisation der Zeit zwischen Sender und Empfänger notwendig, was keine zwingende Verwendung eines Zeitsychronisationsprotokoll erfordert. Obligatorisch ist allerdings die lose Zeitsychronisation, was bedeutet, dass der Empfänger eine obere Grenze der möglichen aktuellen Zeit beim Sender kennen muss. Wie in Abbildung

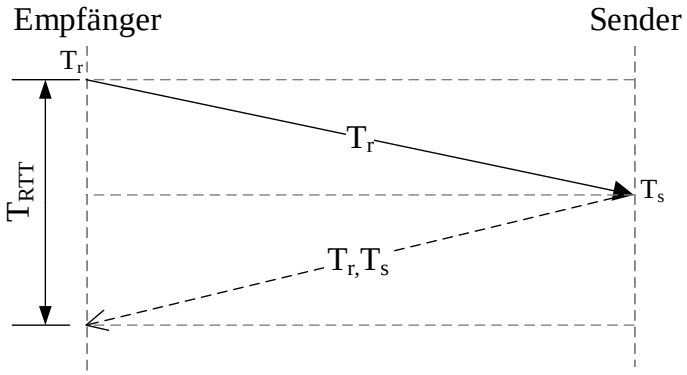

Abbildung 3.1: Zeitsychronisation nach TESLA

3.1 illustriert, initiiert der Empfänger die Zeitsychronisation und schickt dem Sender ein Paket welches die Empfängerzeit T_r enthält. Sobald der Sender ein solches Paket erhält, antwortet er mit einem Paket welches ebenfalls T_r und zusätzlich die eigene Zeit T_s enthält. Mit der beim Empfänger eingehenden Anwort auf die Anfrage zur Zeitsychronisation, kann dieser nun die Zeitabweichung $T_\delta := T_s - T_r$ zur Uhrzeit des Sender sowie die maximale Abweichung T_{ME} berechnen. Dabei gilt $T_{ME} := T_\delta + T_{RTT}$ wobei T_{RTT} die Round-Trip-Time (RTT) ist und beschreibt wie wieviel Zeit es benötigt bis eine Antwort des Senders bei Empfänger vorliegt. Die maximale Zeitabweichung zwischen Sender und Empfänger T_{ME} kann auch als maximaler Fehler der Zeitsychronisation bezeichnet werden.

3.2 One-Way-Key Chains

Viele Applikationen verwenden heutzutage Key-Chains als eine Anreihung von zufälligen Werten, die nicht selten mittels einer Hashfunktion berechnet werden. Hashfunktionen sind Abbildungen einer großen Eingabemenge auf eine meist kleinere Zielmenge und daher nicht injektiv. Abhängig von der konkreten Hashfunktion ist es also möglich für zwei unterschiedliche Eingaben denselben Hashwert geliefert zu bekommen. Solche Kollisionen möchte man jedoch vermeiden und wählt daher eine Hashfunktion die sich dadurch auszeichnet möglichst wenige Kollisionen zu erzeugen. Noch spezifischer sind kryptografische Hashfunktionen, für die zusätzlich gefordert wird, dass es nahezu unmöglich ist Kollisionen willkürlich zu erzeugen. Eine Grundvoraussetzung für Hashfunktionen ist, dass sie nicht surjektiv sind und es mit einem gegebenen Hashwert nicht möglich ist auf den ursprünglichen Eingabewert abzubilden. Dass ist die Eigenschaft weshalb eine Hashfunktion eine Ein-Weg-Funktion darstellt, welche man sich für One-Way-Key Chains zu nutze macht.
Die aufeinander aufbauende Kette von Schlüsseln ist ein wichtiger Bestandteil von TESLA.

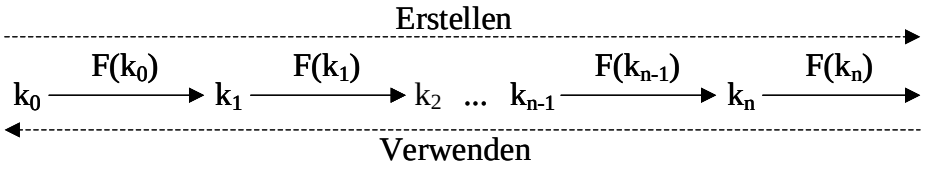

Abbildung 3.2: One-Way-Key Chain

Die Abbildung 3.2 illustiert die Erstellung von Links nach Rechts, wobei F eine rekursiv augerufene kryptografische Hashfunktion bezeichnet. Die Funktion F erhält als Eingabeparameter einen bereits erzeugten Schlüssel k. Intitial kann k_0 ein generierter Schlüssel oder durch eine Pseudo Random Function (PRF) zufällig erzeugter Wert sein. Mit einem solchen k_0, eine kryptografische Hashfunktion F und $N \in \mathbb{N}$, Anzahl der Schlüssel in der Key-Chain (auch Länge der Key-Chain), dann gilt für $n \in \mathbb{N}$ und und $n < N$:

$$k_n = \underbrace{F(...F(F}_{n-mal}(k_0))) = F^n(k_0) \tag{3.1}$$

Analog zur Erzeugung der Key-Chain können einzelne Schlüssel verifiziert werden. Ist k_n bereit ein verifizierter Schlüssel und k_i der i-te Schlüssel aus der Kette, so lässt sich dieser mit

$$k_n = F^n(k_0) = F^{n-i}(F^i(k_0)) = F^{n-i}(k_i) \quad \forall i \in \mathbb{N}; 0 < i < n \tag{3.2}$$

verifizieren. Ist die Gleichung für k_i korrekt, so ist der Schlüssel k_i Teil der Key-Chain, da auch k_n ein Teil der Key-Chain ist. Dieses Verhalten ermöglicht es die erzeugte Key-Chain auf unterschiedliche Arten im Speicher abzulegen.

i) alle berechneten Schlüssel werden gespeichert

ii) nur ein berechneter Schlüssel wird gespeichert

iii) hybride Variante indem nur ein Teil der Schlüssel gespeichert und die Übrigen bei Bedarf berechnet werden

Für alle Variante existieren Vor- und Nachteile bezüglich Speicherbelegung und Rechenzeit. So sind die Zugriffszeiten bei i) zwar gering, es wird jedoch im Vergleich zu iii) mehr Speicher belegt. Wohingegen sich die Zugriffszeiten für einen Schlüssel bei ii) mit größerem n erhöhen.

3.3 Initialisierung Sender

Bevor ein Sender in der Lage ist, selbst-authentifizierende Pakete zu versenden muss er initialisiert werden. Die Sicherheit welche TESLA suggeriert basiert auf Zeitasychronität, weshalb ein zeitreferenzierender Parameter obligatorisch ist. So wird initial die Länge T_{int} des Zeitintervals I sowie ein N, was die Länge der Key-Chain aus Abschnitt 3.2 beschreibt, definiert. So ist T_0 der Startzeitpunkt des ersten Interval I_1. Somit lassen sich alle weiteren Startzeitpunkte mit

$$T_i = T_{i-1} + T_{int} \tag{3.3}$$

sowie der Zeitinterval eines Interval I_i mit

$$I_i = [T_i, T_{i+1}[\tag{3.4}$$

beschreiben. Exemplarisch ist der Startzeitpunkt des zweite Intervall I_2 mit $T_2 = T_1 + T_{int}$ zu berechnen. Die Key-Chain wird mit dem gegebenen N wie im Abschnitt 3.2 beschrieben und verwendet. Wie in Abbildung 3.3 gezeigt wird, wird jedem Interval I_i ein Schlüssel k_{N-i} referenziert, was bedeutet dass dem ersten Interval I_1 der letzte Schlüssel k_{N-1} in der Kette, dem zweiten Interval I_2 der vorletzte Schlüssel k_{N-2} usw. zugewiesen wird. Ein letzter wichtiger Parameter beschreibt, wann ein Schlüssel publiziert wird, damit die Empfänger die erhaltenen Paket verifizieren und gegebenenfalls weiter verarbeiten können. Dieser Parameter ist der *Disclosed Key Interval* und wird im weiteren Verlauf mit d bezeichnet. Damit es möglich ist zu kalkulieren in welchem Interval I_i sich der Sender befindet und somit festzustellen welcher Schlüssel zur Authentifizierung verwendet werden muss und welche Schlüssel veröffentlicht werden darf, muss der Startzeitpunkt T_0 gespeichert werden. Ist der Sender vollständig initialisiert, ist es ihm möglich mit TESLA authentifizierte Pakete zu versenden, sowie allen initializierten Empfänger möglich diese Pakete zu verifizieren.

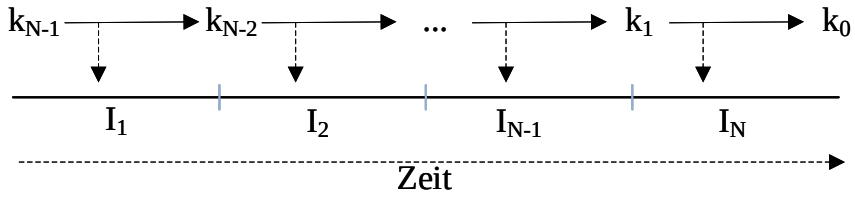

Abbildung 3.3: Schlüsselrefernzierung auf Zeitinterval

3.4 Initialisierung Empfänger

Ebenso wie der Sender, muss jeder Empfänger initialisiert werden bevor er mit TESLA authentifizierte Pakete verifizieren kann. Da die Authentifizierung stark an die Zeit gebunden ist muss der Empfänger zunächst seine lokale Zeit mit dem Sender sychronisieren. Dabei

ist es absolut hinreichend eine lose Zeitsychronisation, wie im Abschnitt 3.1 beschrieben, durchzuführen. Die Abbildung 3.4 zeigt sämliche Kommunikation zur Inititialierung eines Empfängers inklusive der bereits in Abbildung 3.1 gezeigten Schritte zur Zeitsychronisation. Der Empfänger sendet initial ein Paket mit seiner lokalen Zeit T_r und erwartet vom Sender als

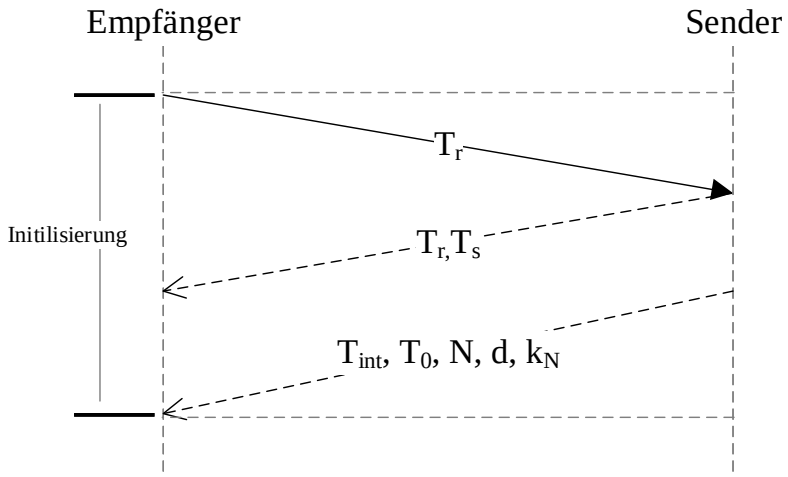

Abbildung 3.4: Initialisierungphase eines Emfpängers

Antwort ein Paket mit T_r und dem Zeitpunkt T_s an welchem der Sender das Paket erhalten hat. Darauf folgend erhält der Empfänger ein weiteres Paket mit allen nötigen TESLA Parametern. Dieses Paket beinhaltet die Länge T_{int} eines Intervals I_i, den Startzeitpunkt T_0 an dem der Sender begonnen hat Daten mit TESLA zu versenden, die Länge der Key-Chain N und schlussendlich die Intervalverzögerung d bis ein Schlüssel veröffentlich wird. Der angehängte Schlüssel k_N dient dem Empfänger, einen zukünftig veröffentlichten Schlüssel zu verifizieren sowie um festzustellen, ob dieser ein valider Schlüssel aus der beim Sender vorliegenden Key-Chain ist.

3.5 Paketauthentifizierung

Sofern Sender und Empfänger vollständig initialisiert sind, können beide Teilnehmer sicher über TESLA kommunizieren. Auf Senderseite besteht die Aufgabe die zu versenden der Daten mit TESLA zu authentifizieren. So muss zunächst das aktuelle Interval I_i anhand der Protokollparameter kalkuliert werden, indem von der aktuelle Senderzeit T_{cur}, T_0 substrahiert und anschließend das Ergebnis mit T_{int} dividiert wird.

$$I_i = \frac{T_{cur} - T_0}{T_{int}} \Longrightarrow T_i = T_0 + i * T_{int} \tag{3.5}$$

Mit dem nun bekannten Interval I_i mit dem Startzeitpunkt T_i kann der Schlüssel k_i berechnet werden, welche zur Berechnung des MAC benötigt wird. Zur Berechnung des MAC wird eine spezielle Hashfunktion F' verwendet, welche im Gegensatz zu einfachen Hashfunktion F, zwei Parameter benötigt. Die Parameter der Funktion F' setzen sich zum Einen aus einem kryptografischen Schlüssel und zum Anderen aus den Daten, für die ein MAC berechnet werden soll, zusammen. Das Ergebnis von F' ist ein *Keyed-Hash Message Authentication*

Code [19] (HMAC). So wird der HMAC, welche für die Authentifizierung benötigt wird, mit den zu sendenden Daten M sowie dem Schlüssel k_i für das aktuelle Interval I_i wie folgt berechnet.

$$k'_i = F'(k_i, M) = HMAC_M \tag{3.6}$$

Zur Authentifizierung von, in vorangegangenen Intervallen versendeten Paketen ist es notwenig einen Schlüssel zu berechnen der veröffentlicht werden soll. Für die Berechnung des zu veröffentlichen Schlüssel k_{pub}, wird unter Berücksichtigung der initial festgelegten Intervallverzögerung d der Index ausgehend des aktuellen Intervall I_i kalkuliert.

$$k_{pub} = F^{i-d}(k_0) \quad \forall i \in \mathbb{N}; i < d \tag{3.7}$$

Anschließend kann der Sender ein mit TESLA authentifiziertes Paket P zusammenbauen. Die Authentifizierungsinformation werden mit den Daten M konkatiniert und beinhalten alle nötigen Information, welche beim Empfänger zur Verifizierung und Authentifizierung obligatorisch sind.

$$P_m = \{M_m||i||k'_i||k_{pub}\} \quad \forall m \in \mathbb{N} \tag{3.8}$$

Das geschiet für alle Daten M welche in dem entsprechenden Intervall I_i versenden werden, wie die Abbildung 3.5 zeigt. Erhält ein Empfänger Pakete muss dieser zunächst prüfen ob das

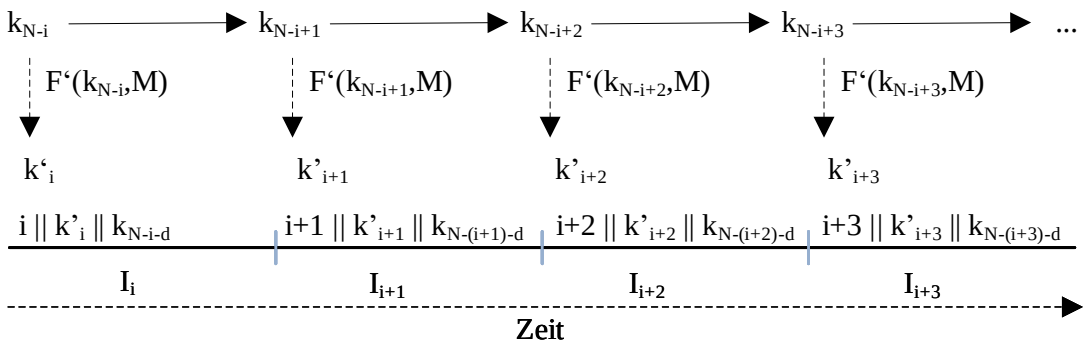

Abbildung 3.5: Erzeugung von Authentifizierungsdaten

im Paket enthaltene Interval i_P plausibel ist und auch in diesem versandt worden ist. Dafür wird ähnlich wie beim Sender zunächst der aktuelle angenommene Intervall i_{kum} kumuliert. Die Formel welche zur Berechnung des Intervalls verwendet wird, unterscheided sich lediglich darin, dass zusätzlich die maximale Zeitabweichung T_{ME} aus der Zeitsychronisierung mit einfließt.

$$i_{kum} = \frac{T_{cur} - (T_0 + T_{ME})}{T_{int}} \tag{3.9}$$

Das angenommen Intervall i_{kum}, in welchem sich der Sender maximal befinden kann, wird weiter zu Validierung verwenden. Ist der im Paket P_m enthaltenen Interval $i_P < i_{kum}$, wird das Paket angenommen und zwischengespeichert bis der dazugehörige Schlüssel k_i veröffentlicht wird und vom Empfänger authentifiziert werden kann. Anschließend wird der ebenfalls im Paket P_m enthaltene veröffentliche Schlüssel k_{pub} validiert. So muss die Gleichung 3.10 *WAHR* sein damit der Schlüssel akzeptiert wird und zur authentifizierung bereits

gepufferten Pakete genutzt werden kann.

$$k_N \stackrel{!}{=} F^{i_{kum}}(k_{pub}) \tag{3.10}$$

Dabei ist k_N der Schlüssel welchen der Empfänger in der Initialisierungphase erhalten hat, k_{pub} der im Paket P_m enthaltene veröffentliche Schlüssel und M_m die Daten welche das Paket P_m beinhaltet. Alternativ zur Gleichung 3.10 kann die Validierung des Schlüssels k_{pub} auch wenig aufwendig überprüft werden, sofern ein bereits veröffentlicher Schlüssel k_v vorliegt. Dabei ist der Schlüssel k_v ein bereits veröffentlicher Schlüssel, welcher dem Intervall I_v zugewiesen ist und bereits erfolgreich validiert ist.

$$k_v \stackrel{!}{=} F^{i_{kum}-v}(k_{pub}) \tag{3.11}$$

Kommt der Empfänger zu dem Ergebnis, dass der Schlüssel k_{pub} valide ist, wird dieser genutzt um die zwischengespeicherten Pakete zu authentifizieren. Der Empfänger iteriert über alle Pakete und überprüft für jedes Paket zunächst anhand des Paketintervalls i_P ob es authentifiziert werden kann.

$$i_p \leq i_{kum} - d \tag{3.12}$$

Kann das Paket anhand dieser Überprüfung potenziell authentifiziert werden, wird der entsprechende Schlüssel analog zur Formel 3.2 berechnet womit der HMAC erzeugt werden kann. Dabei entspricht n dem referenzierden Index I_{pub} von k_{pub}, i dem Intervall i_P des zu authentifizierden Pakets und der Funktionparameter k_i dem veröffentlichen Schlüssel k_{pub}. Anschließend kann der im Paket P_m hinterlegte HMAC k'_i mit dem zuvor Berechneten verglichen werden, wobei gelten muss.

$$k'_i \stackrel{!}{=} F'(k_n, M_m) \quad mit \quad k_n \stackrel{3.2}{=} F^{I_{pub}-i_P}(k_{pub}) \tag{3.13}$$

Ist diese Gleichung korrekt, ist das Paket erfolgreich authentifiziert, kann aus dem Zwischenspeicher oder auch Puffer entfernt und weiter verarbeitet werden. Schlägt diese Überprüfung fehl, wird das Paket verworfen und ebenfalls aus dem Paketpuffer entfernt.

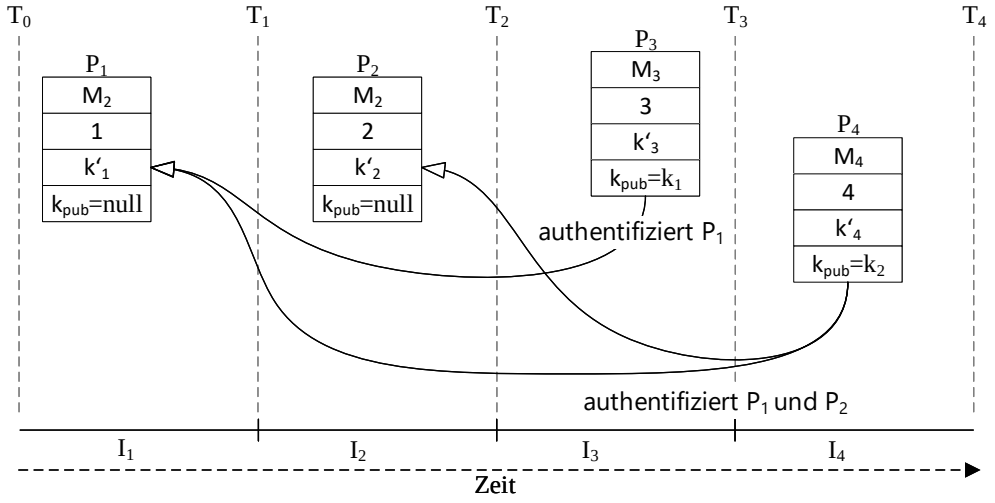

Abbildung 3.6: Beispiel zur Authentifizierung von Paketen für d = 2

4 Implementierung

Nachdem die erforderlichen Grundlagen geklärt wurden, wird im folgenden Kaptiel näher beschrieben wie das unter Abschnitt 3 erklärte Protokoll TESLA prototypisch implementiert wurde. Die gewählte Programmiersprache ist hierbei C, da es neben C++ vollumfänglich von dem Betriebsystem Riot-OS, wie im Kapitel 2.3 ausführlicher aufgeführt, unterstützt wird. Auf Grund der heterogenen Geräteunterstützung und der Möglichkeit auf kleinsten Geräte (siehe auch Tabelle 2.2) zu laufen, wurde Riot-OS als Ausgangsbasis für die im Folgenden beschriebene programmatische Umsetzung gewählt. TESLA stellt die Anforderung, dass die Initialisierung der Kommunikationsteilnehmer über einen sicheren Kanal abzuwickeln ist, weshalb das Transportsicherheitsprotokoll ESP 2.1.3 als Rahmen für alle IP-Pakete dient. ESP ist ein bereits etabliertes Protokoll und bietet einen Mechanismus für einen sicheren Austausch von Daten zwischen Kommunikationsendpunkten. Dieser Mechanismus wird um die TESLA Kernfunktionalität erweitert, wodurch TESLA vollumfänglich in ESP eingebettet wird und die darin spezifizierten Verfahren verwenden werden können. Ein wichtiger adaptierbarer Bestandteil von IPsec ist der Austausch von Schlüsselmaterial mit IKE (siehe Kapitel 2.1.4), welche im Rahmen dieser Arbeit nicht weiter betrachtet wird. Da der Austausch von Schlüsseln nicht im Fokus dieser Arbeit steht, wird für die initiale Authentifizierung ein PSK verwendet. Es ist jedoch möglich für zukünftige Anwendungen diesen gegen ein alternatives Verfahren auszutauschen. Des weiteren sieht IPsec vor, dass die zu verwendeten kryptografischen Algorithmen zwischen beiden Endpunkten ausgehandelt werden. Riot-OS bietet verschiedene Implementierungen (MD5, SHA-1, SHA-256 und SHA-3) zur Berechnung eines Hashwertes an, wobei MD5 und SHA-1 unlängst als unsicher eingestuft werden. Die Wahl zwischen SHA-256 und SHA-3 fiel auf Grund der bereits vorhandenen Funktionalität auf SHA-256. Die SHA-256 Implementierung in Riot-OS bietet zusätzlich noch das Erzeugen einer Key-Chain, welche für die Protokollimplementierung genutzt werden kann. Im weiteren Verlauf gilt somit die kryptografische Hashfunktion SHA-256 als bereits ausgehandelter Algorithmus für die Berechnung der Authentifizierungsinformation.

4.1 RIOT Datenstrukturen

Für das Versenden und Empfangen von Netzwerkpaketen bieten Riot-OS die abtrakte Schnittstelle *Generic Network Stack (GNRC)* an. Der GNRC speichert alle ein- und ausgehenden Pakete in einem zentralen Paketspeicher *pktbuf* zwischen, wobei die einzelnen Nachrichten als verkettete Liste organisiert werden. Die Elemente dieser Listen sind Datenblöcke (*Snips*) verschiedener Größe und je einzeln verarbeitbar. Typischerweise repräsentieren diese *Snips* eine Payload oder einen Protokoll-Header. Den generellen Aufbau eines *Snips* definiert Riot-OS in der C-Struktur *gnrc_pktsnip_t*, welche einen Pointer auf die Daten (data), die Länge der Daten (length), einen Typen (type) und einen Pointer (next) auf ein eventuell existierendes nachfolgendes Paket beinhaltet. Zur internen Speicherverwaltung dient das Feld *users*, welches ebenfalls ein Teil von der C-Struktur ist und beschreibt ob der referenzierte Speicher freigegeben werden kann sobald das Feld auf den Wert 0 gesetzt ist.

Die Abbildung 4.1 illustiert beispielhaft den Aufbau einer Nachricht im *pktbuf* anhand eines UDP Pakets. Dabei entspricht prinzipiell jedes der *Snips* einem Teil eines IP-Datagrams und verdeutlich wie eine Kette von *Snips* zu einem vollständigem IP-Paket zusammengesetzt werden kann. Das Erstellen eines neues *Snips* sollte über den Aufruf der Funktion

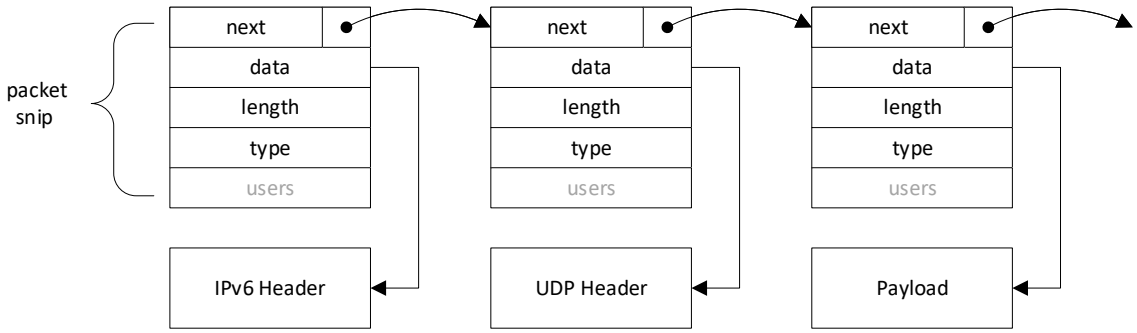

Abbildung 4.1: GNRC Paketsnips im Paketspeicher am Beispiel UDP

gnrc_pktbuf_add(next,data,size,type) erfolgen. Somit ist gewährleistet das Riot-OS den allokierten Speicher für das neue *gnrc_pktsnip_t* verwaltet und keine Speicherlöcher entstehen, in dem nicht mehr benötigter Speicher durch das System wieder frei gegeben wird. Steht ausreichend Speicher zur Verfügung um ein neues *Snip*, mit der durch *size* angegebenen Größe, zu erzeugen gibt die Funktion *gnrc_pktbuf_add(*next,*data,size,type)* einen Pointer zu dem neu erzeugtem *Snip* zurück. Wird ein *Snip* nicht mehr benötigt kann mit der Funktion *grnc_pktbuf_release(*pktsnip)* das Feld *users* um eins dekrementiert werden. Da zunächst alle bei einem Host eingehende Nachrichten aus nur einem *Snip* bestehen und sämtliche Daten enthält muss dieses zur Weiterverarbeitung in den ursprünglichen Zustand versetzt werden. Um die Daten für die entsprechende weitere Verarbeitung aufzuteilen stellt der GNRC die Funktion *gnrc_pktbuf_mark(*pktsnip, size,type)* bereit. Diese erzeugt intern ein neues *Snip* vom als Parameter angegebenen Typen, befüllt es entsprechend des übergebenen Parameters und gibt es zurück. Somit ist der Parameter *length* gleich dem Wert im Feld *size* und das Feld *data* zeigt auf die Speicheradresse, an welcher die Daten mit der Länge *size* abgelegt sind (siehe Abbildung 4.2). Außerdem wird der Pointer *data* im übergebenen *Snip* angepasst in dem es auf die Speicheradresse der verbleibenden noch nicht markierten Daten zeigt. Dies kann wiederholt aufgerufen werden bis alle Daten in dem ursprünglichen *Snip* einem neuem *Snip* mit dem entsprechenden Typen zugewiesen worden sind.

4.2 ESP in RIOT OS

Riot-OS bietet zum Entstehungszeitpunkt dieser Arbeit keine Implementierung von ESP im Kernel an, weshalb es erforderlich ist ESP zu implementierten um es für die Umsetzung des TESLA Protokolls verwenden zu können, jedoch nicht vollumfänglich in den GNRC von Riot-OS zu integrieren. Die vollständige Integration von ESP in den Kernel von **riots!** ist Bestandteil der Arbeit *Diet-ESP for RIOT OS* [21] von Max Malkus, die zum aktuellen Zeitpunkt nicht abgeschlossen ist.
Wie die Abbildung 4.3 zeigt, sind die zu übertragenen Nutzdaten vollständig in dem ESP Header inkludiert, wodurch es nicht möglich ist eine C-Struktur zu definieren, welche alle

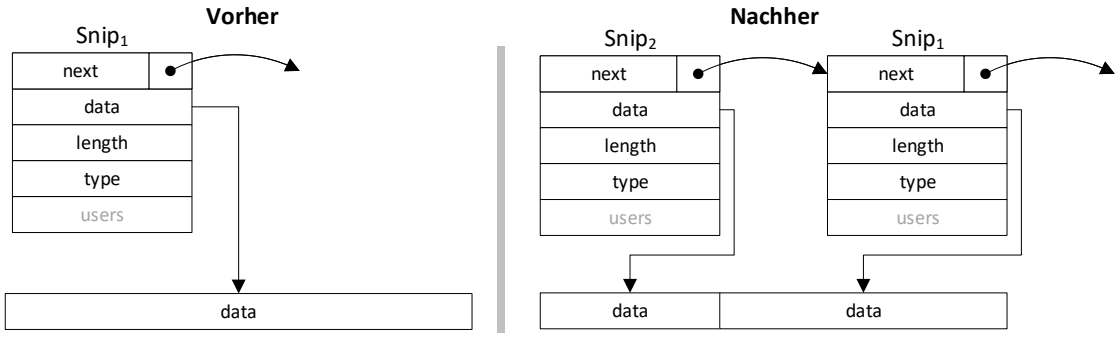

Abbildung 4.2: Markierung von GNRC *Snips* im Paketspeicher

Felder des Headers umfasst. Zusätzlich kann die eingeschlossene Payload von variabler Länge sein, was die Definition einer C-Struktur mit statischen Größen unmöglich macht. Aus diesem Grund ist es notwendig den Header auf verschiedenen C-Strukturen aufzuteilen, welches je auf ein *Snip* abgebilden werden kann. So ist der Header auf drei C-Strukturen aufgeteilt, wobei die Payload bedingt durch die variable Länge separat betrachtet wird. Der Codeausschnitt 4.1 zeigt den ersten Teil und beinhaltet die Felder SPI *spi* und die Sequenznummer *seq_num*.

```
typedef struct __attribute__((packed)) {
    /* security parameters index (SPI) of this packet. */
    uint32_t spi;
    /* sequence number of this packet. Increases by one for each packet sent */
    uint32_t seq_num;
} ipv6_ext_esp_header_hdr_t;
```

Listing 4.1: C-Struct ESP Header

Die SPI ist eine abiträre Zahl, welche zur Identifizierung eines Eintrags in der SAD dient. Der konkrete Wert wird von jedem Sender willkürlich bestimmt, wodurch Kollisionen nicht ausgeschlossen werden können. Aus diesem Grund kann, wie im Abschnitt 2.1.1 beschrieben, zusätzlich mittels der Quell- und wenn nötig auch der Zieladresse ein unikaler Eintrag in der SAD gefunden werden. Da die möglichen Werte welche die SPI annehmen kann, keinen bestimmten Einschränkungen unterliegen, werden in diesem Prototypen die verwendeten SPIs auf den Wert der Quelladresse gesetzt. Die Bedeutung der Sequenznummer *seq_num* ist offensichtlich trivial und wird für jedes ausgehende Paket um eins erhöht, wodurch eine Paketreihenfolge implizit gegeben ist.

```
typedef struct __attribute__((packed)) {
    /* padding length of this packet. */
    uint8_t pad_len;
    /* type of next header in this packet. */
    uint8_t nh;
} ipv6_ext_esp_payload_trailer_hdr_t;
```

Listing 4.2: C-Struct ESP Trailer

Die auf die Payload folgenden Felder Padding-Länge *pad_len* und Next-Header *nh* beinhaltet die C-Struktur im Codeausschnitt 4.2, welche den zweiten Teil des aufgeteilten ESP Headers

repräsentiert. Das Auffüllen von Bytes (Padding) der Payload auf eine bestimmte Blockgröße ist motiviert durch zwei Anwendungsfällen. i) Bei der Verwendung von Verschlüsselung verlangen einige Algorithmen, dass die Länge der zu verschlüsselnden Daten ein Vielfaches einer bestimmten Anzahl von Bytes ist, um zum Beispiel der geforderten Blockgröße einer Cipher Block Chaining (CBC) Verschlüsselung zu entsprechen. ii) Ungeachtet der Anforderungen eines Verschlüsselungsalgorithmus, muss garantiert werden, dass die Felder *pad_len* und *nh* in einem 2-Byte Block rechtsbündig positioniert sind, wie es die Abbildung 4.3 zeigt. Ist das Auffüllen von mit Nullen beschriebenen Bytes (0-Bytes) notwenig, so gibt das Feld *pad_len* an wieviel Bytes mit der Payload konkatiniert werden, damit der Empfänger in der Lage ist den Ursprungszustand der Payload wiederherzustellen. Das Feld Next-Header *nh*, gibt an von welchem Typ der in Payload stehende Header ist. Beinhaltet die Payload kein spezifiziertes Protokoll, sondern reine Daten, beschreibt *nh* mit dem Wert 59 (*No Next Header*), dass kein weiterer Header eines Protokolls folgt. Der letzte und somit noch ausstehenden Teil des

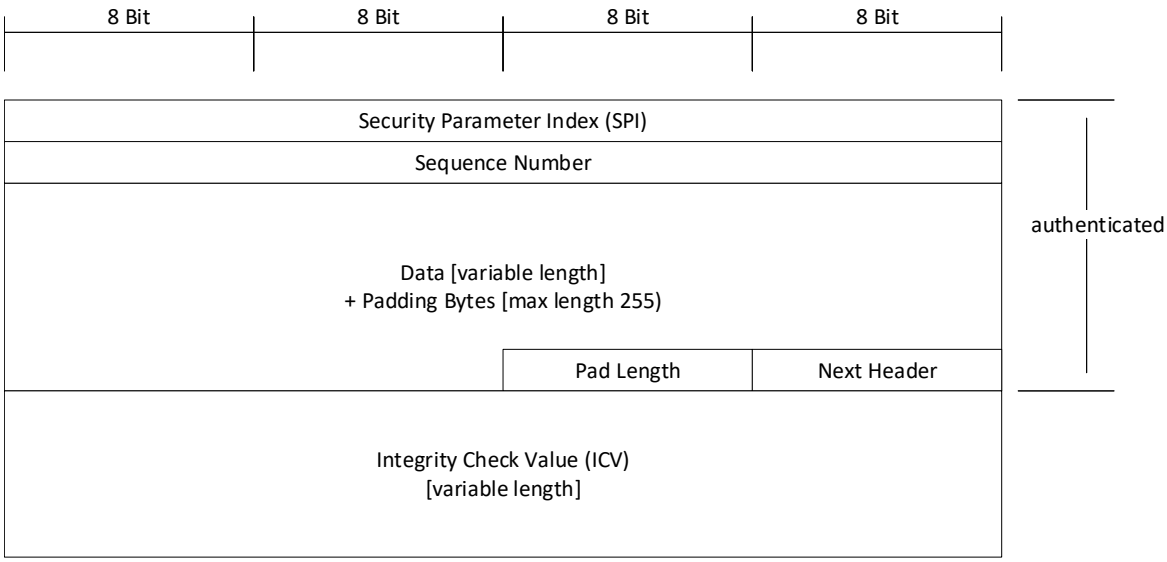

Abbildung 4.3: ESP Header

ESP Headers, den ICV, beschreibt die C-Struktur im Codeausschnitt 4.3. Der ICV besteht aus dem Feld MAC *mac*, welches die berechneten Authentifizierungsinformation enthält und der Länge des Byte-Arrays *macLength*. Da in dieser Implementierung der kryptografische Algorithmus SHA-256 verwendet wird, ist folglich die Konstante *KEY_LENGTH* mit dem Wert 32 belegt, wodurch die Länge des Byte-Arrays *mac* ebenfalls 32 ist.

```
1 typedef struct __attribute__ ((packed)) {
2     unsigned char mac[KEY_LENGTH];
3     uint16_t macLength;
4 } ipv6_ext_esp_icv_hdr_t;
```

Listing 4.3: C-Struct ESP ICV

4.2.1 ESP Paketerstellung

Da der ESP Header auf mehrere C-Strukturen aufgeteilt ist, ist es notwendig diese beim Erzeugen eines ESP Pakets wieder zusammenzufügen. Dies erfolgt mittels der in Codeausschnitt 4.4 gezeigten Funktion *gnrc_ipv6_esp_build(*payload,*sa)*, die ein *Snip* als Payload und eine SA als Parameter annimmt. Bedingt durch die vom GNRC vorgegeben Struktur müssen die einzelnen *Snips* in der Reihenfolge erzeugt werden, wie es im ESP Header spezifiert ist um ein valides Paket zu erhalten. Folglich wird zunächst ein *Snip* für SPI und Sequenznummer erzeugt und beschrieben. Anschließend erfolgt das eventuell nötige Byte-Padding, was eine Modifikation der Payload impliziert sowie das Erzeugen des *Snips* für die Padding-Länge und den Next Header. Final wird mit der Funktion *gnrc_pkt_esp_build(*header, *payload, *trailer)* das ESP Paket exklusiv des ICV, welcher später berechnet wird, in der korrekten Reihenfolge zusammengesetzt und zurückgegeben.

```
1  gnrc_pktsnip_t *gnrc_ipv6_esp_build(gnrc_pktsnip_t *payload, sa_t *sa) {
2      sa->senders_seq_num++;
3      network_uint32_t ip = sa->senders_ip.u32[4];
4      gnrc_pktsnip_t *header = ipv6_esp_header_build(
5                                                  ip.u32, sa->senders_seq_num);
6      if (header == NULL) {
7          LOG_ERROR("Error on build esp header packet\n");
8          return NULL;
9      }
10     // add padding if neccesary
11     int pad_len = esp_add_padding(payload);
12     if (pad_len == -1) {
13         LOG_ERROR("Error on add payload padding!\n");
14         return NULL;
15     }
16     gnrc_pktsnip_t *trailer = ipv6_esp_payload_trailer_build(
17                                          (uint8_t) pad_len, PROTNUM_RESERVED);
18     if (header == trailer) {
19         LOG_ERROR("Error on build esp trailer packet\n");
20         return NULL;
21     }
22     gnrc_pktsnip_t *esp = gnrc_pkt_esp_build(header, payload, trailer);
23     return esp;
24 }
```

Listing 4.4: C-Funktion zur Erstellung eines ESP Pakets

4.2.2 Berechnung des ICV

Mit dem Aufruf von *gnrc_ipv6_esp_build(*payload,*sa)* erhält der Aufrufer zwar ein korrektes ESP Paket, jedoch entspricht es in diesem Zustand noch nicht vollständig der Spezifikation. Diese verbietet, dass weder Verschlüsselung noch Authentifizierung unbenutzt bleiben dürfen. Da im Rahmer dieser hier beschriebenen Implementierung keine Verschlüsselung der Payload statt findet, ist folglich die Berechnung eines ICV erforderlich. Die Berechnung des ICVs erfolgt mit der Funktion *gnrc_ipv6_esp_calc_icv(*esp,*key,key_size)* (siehe Codeausschnitt 4.5) unter der Hereingabe des ESP Pakets *esp* für welches der ICV berechnet werden soll, sowie eines kryptografischen Schlüssels *key* und dessen Länge *key_size*. In dieser wird zunächst ein *Snip* für die C-Struktur (siehe Codeausschnitt 4.3) erzeugt um den mit

der Funktion *hmac(*key, key_size, *esp→data, esp→size, *icv-mac)* erzeugtem HMAC und dessen Länge speichern zu können. Abschließend wird das neu erzeugte *Snip* an das hereingegebene *Snip* angehängt. An diesem Punkt entspricht die Kette aus *Snips* einem validen ESP Paket und kann versendet werden.

```
gnrc_pktsnip_t *gnrc_ipv6_esp_calc_icv(gnrc_pktsnip_t *esp, void *key,
                                       size_t key_size) {
    gnrc_pktsnip_t *pkt = gnrc_pktbuf_add(NULL, NULL,
                                          sizeof(ipv6_ext_esp_icv_hdr_t),
                                          GNRC_NETTYPE_UNDEF);
    if (pkt == NULL) {
        LOG_ERROR("Error on build icv packet\n");
        return NULL;
    }
    ipv6_ext_esp_icv_hdr_t *icv = pkt->data;
    icv->macLength = KEY_LENGTH;
    hmac(key, key_size, esp->data, esp->size, icv->mac);
    LL_APPEND(esp, pkt);
    return pkt;
}
```

Listing 4.5: C-Funktion zur Berechnung des ICVs

4.3 Nachrichtenempfang

Da ESP ein Teil von IPv6 darstellt, erwartet der in dieser Arbeit beschriebene Prototyp eingehende IP-Pakete. Das hat zur Folge, dass auch Pakete verarbeitet werden müssen, die keine für den Prototypen relavanten Information enthalten. Dies sind vorrangig Internet Control Message Protocol (ICMP) [5] Pakete, die zum Austausch von Fehler- und Informationsmeldungen sowie bei der Verwendung von IPv6 der Neighbor Discovery [25] dienen. Der Codeausschnitt 4.6 zeigt, wie die initiale Behandlung von eingehenden IP-Paketen erfolgt. Da grundlegend ICMP Pakete für diese Implementierung irrelevant sind, werden diese, nachdem das eingegangene IP-Paket mittels der Funktion *handle_pkt_ip(*pkt)* entsprechend der C-Strukur *ipv6_hdr_t* markiert wurde (siehe 4.2), anhand des Feldes Next Header *nh* herausgefiltert und verworfen.

```
gnrc_pktsnip_t *handle_incomming(gnrc_pktsnip_t *pkt) {
    gnrc_pktsnip_t *ip = handle_pkt_ip(pkt);
    if(ip == NULL){
        LOG_ERROR("Cannot handle ip pkt\n");
        return NULL;
    }
    ipv6_hdr_t *hdr = ip->data;
    if (hdr->nh == PROTNUM_ICMPV6) {
        LOG_DEBUG("Ignore ICMPv6 Packages\n");
        return NULL;
    }
    return ip;
}
```

Listing 4.6: C-Funktion zur behandlung von eingehenden Paketen

Die Funktion *handle_incomming(*pkt)* kommt wiederum in einer weiteren Funktion zur Anwendung wie der Codeausschnitt 4.7 zeigt. Da kontinuierlich Nachrichten empfangen werden sollen, wird die vom GNRC Stack bereitstellten Funktion *msg_receive* in einer Endlosschleife aufgerufen. Außerdem reagiert diese Funktion auf Nachrichten vom Typ *TESLA_IPC_STOP_RECV*, die eine Abbruchbedingung definiert, wodurch die Endlosschleife beendet werden kann. Ist die eingegangene Nachricht vom Typ *GNRC_NETAPI_MSG_TYPE_RCV* und konnte als ein relevantes IP-Paket identifiziert werden, wird zur weiteren Verarbeitung die Funktion *pHandleMsgFunc(*pkt, *sa)* aufgerufen. Diese Funktion ist zum Zeitpunkt des Kompilierens unbekannt, da der Parameter von *rcv_loop* lediglich einen Zeiger auf eine Funktion annimmt.

```
1  void rcv_loop(void (*pHandleMsgFunc)(gnrc_pktsnip_t *pkt, sa_t *sa)) {
2      gnrc_pktsnip_t *ipPkt;
3      msg_t msg;
4      int run = 1;
5      while (run == 1) {
6          msg_receive(&msg);
7          gnrc_pktsnip_t *pkt = msg.content.ptr;
8          if (msg.type == TESLA_IPC_STOP_RECV) {
9              run = 0;
10             continue;
11         }
12         if (msg.type == GNRC_NETAPI_MSG_TYPE_RCV) {
13             ipPkt = handle_incomming(pkt);
14             if (ipPkt != NULL) {
15                 ipv6_addr_t *ip_addr = grnc_pkt_ip_src(ipPkt);
16                 sa_t *sa = sa_by_ip(ip_addr);
17                 pHandleMsgFunc(pkt, sa);
18             } else {
19                 LOG_ERROR("no tesla msg\n");
20             }
21         } else {
22             LOG_DEBUG("received unidentified message\n");
23         }
24     }
25     deregisterServerBootstrap();
26     LOG_INFO("receiving thread stopped!");
27 }
```

Listing 4.7: C-Funktion zum wiederholten Empfangen von Nachrichten

Die Hereingabe eines Funktions-Pointer lässt sich in diesem konkrete Szenario mit einer Wiederverwendung begründen, wodurch Sender und Empfänger die selbe Funktion *rcv_loop* verwenden können, es allerdings möglich ist die eingegangen Nachrichten jeweils in unterschiedlichen Funktionen weiter zu verarbeiten. Weil die Funktion *msg_receive* blockiert bis eine neue Nachricht vorhanden ist und somit auch die Funktion *rcv_loop* blockiert, wird diese in einem seperaten Thread aufgerufen, um zu verhindern, dass die gesammte Applikation ebenfalls blockiert wird.

4.4 TESLA

Nachdem gezeigt worden ist, wie grundlegend IP-Pakete empfangen, gefiltert und für die weitere Verarbeitung vorbereitet werden, wird im Folgenden zunächst auf die Implementierung zur Erzeugung einer Key-Chain eingegangen. Außerdem wird erläutert, wie der Benutzer einen Schlüssel abfragen und verifizieren kann. Dabei gilt, wie bereits erwähnt, die kryptografische Hashfunktion SHA-256 des Riot-OS Kernels als bereits ausgehandelt. Anschließend daran wird die Weiterverarbeitung von eingehenden Pakete während der Initialisierungsphase (siehe Kapitel 3.3) auf Seiten des Empfängers sowie des Senders beschrieben.

4.4.1 Key-Chain

Die Erstellung der Key-Chain erfolgt mit der Funktion *generateKeyChain(len, wp_len, *seed)*, welche die Parameter, Länge der Key-Chain *len*, Anzahl an Wegpunkten *wp_len* und den initiale Eingabewert *seed* für die Hashfunktion F' annimmt. Der Parameter *seed* ist ein Byteblock mit der Länge, welche die Konstante *KEY_LENGTH* definiert und typischerweise mit zufälligen Werten aus einer PRF beschrieben. Wie im Abschnitt 3.2 bereits erwähnt, existieren verschiedene Optionen zur Speicherung der Key-Chain die jeweils speicher- oder laufzeitoptimiert sind. Wieviel Gewicht einer der beiden Optimierungsoptionen zugesprochen werden soll, kann mit der Anzahl von Wegpunkten *wp_len* für die zu erzeugende Key-Chain gesteuert werden. Ist der Parameter *wp_len* auf einen Wert gesetzt, dass $len =< wp\_len$ gilt, werden alle erzeugten Schlüssel gespeichert. Gilt dies nicht, wird jeder n-te Schlüssel als Wegpunkt gespeichert, wobei n mit $floor(len/wp\_len)$ berechnet wird. Wie das Aussehen kann, zeigt die Abbildung 4.4 beispielhaft für $n = 2$. Der Abruf eines Schlüssels erfolgt mit der Funktion *getKeyWithIndex(index,*key)*, welche einen Index sowie einen Pointer auf eine Speicheradresse annimmt, die angibt wo der Schlüssel im Speicher abgelegt werden soll. Zur Verifikation ob ein Schlüssel Teil der erzeugten Key-Chain ist, steht die Funktion *verifyKey(index,*key)* zur Verfügung, welche ebenfalls einen Index sowie einen Pointer auf den zu verifizierenden Schlüssel als Parameter akzeptiert. Wird ein Schlüssel erfolgreich als Teil der Key-Chain identifiziert, so gibt die Funktion 0 zurück, andernfalls 1.

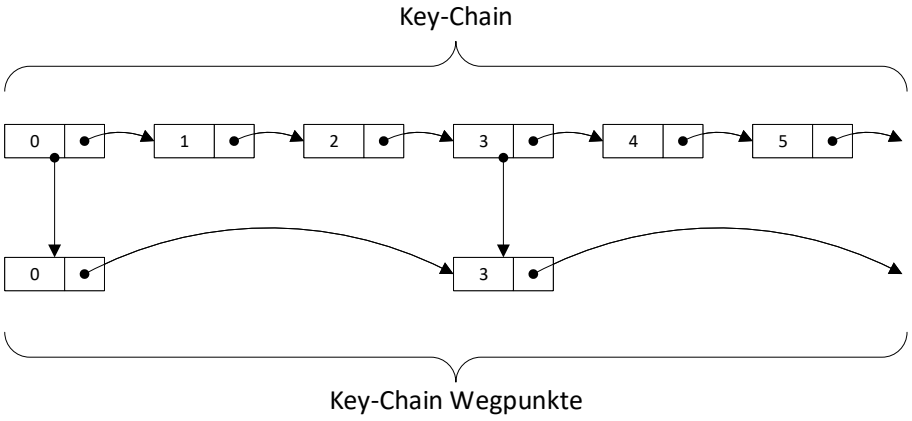

Abbildung 4.4: One-Way-Key Chain mit Wegpunkten

4.4.2 Initialisierung

Die Initialisierung von Empfängern beinhaltet, wie in Kapitel 3.4 erläutert, grundlegend zwei Schritte. Da der Aufbau eines sicheren Kanals nicht Bestandteil von TESLA ist, wird darauf nicht weiter eingegangen und als gegeben betrachten. Die zwei TESLA relevanten Schritte lassen sich in die Zeitsychronisation und den Erhalt der vom Sender festgelegten TESLA Konfigurationsparameter (siehe Codeausschnitt 4.8) unterteilen, wie es das Kapitel 3.1 beschreibt.

```
typedef struct {
    uint32_t intervalDuration;
    uint32_t startTime;
    uint32_t keyChainLength;
    uint16_t keyDisclosureDelay;
    uint16_t keyLength;
    unsigned char keyData[KEY_LENGTH];
} __attribute__ ((packed)) tesla_config_t;
```

Listing 4.8: C-Struktur für Konfigurationsparameter

Der finale Status nach erfolgreicher Clientinitialisierung ist *TESLA_DATA* und signalisiert, dass der Client bereit zum Empfang von mit TESLA authentifizierten Paketen ist. Die verschiedenen Phasen des TESLA Verbindungsstatus sind in der Enumeration *tesla_phase* abgebildet und finden in der Funktion *handle_msg_client*, welche im Codeausschnitt 4.9 gezeigt ist, ihre Verwendung. Da die empfangenen Nachrichten in der jeweiligen Phase unterschiedlich weiterverarbeitet werden, muss abhängig von der Phase die entsprechend Funktion aufgerufen werden, um das Datenpaket korrekt verarbeiten zu können. So wird in der Phase der Zeitsychronisation *TESLA_TIME_SYNC*, die maximale Zeitabweichung T_{ME} berechnet und in einer dafür definierten C-Struktur gespeichert. Ist der aktuelle Verbindunsstatus auf *TESLA_BOOTSTRAP* gesetzt, so wird eine Nachrichten erwartet, welche alle für den Client notwendigen TESLA Parameter enthält. Darunter auch ein Schlüssel k, gegen welchen alle zukünftig veröffentlichen Schlüssel verifiziert werden können.

```
void handle_msg_client(gnrc_pktsnip_t *pkt, sa_t *sa) {
    switch (clientRunConfig.phase) {
        case TESLA_TIME_SYNC:
            handleTimeSyncResponsePacket(pkt, sa);
            break;
        case TESLA_BOOTSTRAP:
            handleBootstrapConfigResponsePacket(pkt, sa) == 0)
            break;
        case TESLA_DATA:
            handle_pkt_tesla(pkt, sa);
            break;
        default:
            LOG_ERROR("PHASE: Unknown! Ignore package");
    }
}
```

Listing 4.9: C-Funktion zur Nachrichtverarbeitung beim Empfänger

Wie zuvor im Abschnitt 4.3 erwähnt nimmt die im Codeausschnitt 4.7 gezeigte Funktion einen Funktions-Pointer entgegen, um für Sender und Empfänger wiederverwendet werden zu

können. Die Funktion *handle_msg_client(*pkt, *sa)* stellt eine Möglichkeit dar, wie die Pakete weiterverarbeitet werden können und entspricht der Form, welche in der Funktion *rcv_loop* als Parameter definiert ist. Analog zum Empfänger müssen eingehende Pakete beim Sender weiterverarbeitet werden, weshalb ebenfalls eine Funktion (siehe Codeausschnitt 4.10), mit der in der Funktion *rcv_loop* vorgegebenen Form, zur Verfügung steht. Da ein Empfänger die Initialisierung mittels einer Anfrage zur Zeitsychronisation einleitet, reagiert der Sender, ausschließlich auf Pakete während sich ein Empfänger in der Phase *TESLA_TIME_SYNC* befindet. Entsprechend der unter 3.1 beschriebenen Sequenz zur Zeitsychronisierung, extrahiert der Sender die im Paket gesendete Zeit mit der Funktion *handleTimeSyncRequestPacket(*pkt, *sa)*, in welcher außerdem die Validierung des ESP Headers stattfindet. Anschließend antwortet der Sender mit einem neu erzeugtem Paket zur Zeitsychronisierung, welches er mit der empfangen Zeit des Clients *client_time* und der eigenen Zeit beschreibt. Darauffolgend wird ein weiteres Paket erzeugt, in dem die für Client notwenigen Konfigurationsparameter (siehe Codeausschnitt 4.8) enthalten sind. Da sämtlicher Datenaustausch zwischen Sender und Empfänger über einen sicheren Kanal stattfinden muss, werden die Pakete, wie der Abschnitt 4.2.1 beschreibt, mit ESP sicher übertragen.

```
void handle_msg_srv(gnrc_pktsnip_t *pkt, sa_t *sa) {
    gnrc_pktsnip_t *responsePayloadPktSnip = NULL;
    switch (sa->phase) {
        case TESLA_TIME_SYNC:
            int32_t client_time = handleTimeSyncRequestPacket(pkt, sa);
            if(client_time > 0) {
                sa->phase = TESLA_BOOTSTRAP;
                // send time sync response
                responsePayloadPktSnip =
                  buildTimeSyncResponse((uint32_t) client_time);
                responsePayloadPktSnip =
                  gnrc_ipv6_esp_build(responsePayloadPktSnip, sa);
                gnrc_ipv6_esp_calc_icv(responsePayloadPktSnip, sa->sharedKey,
                                       sa->sharedKeyLength);
                send(responsePayloadPktSnip, &ip_addr);
                // send tesla config
                responsePayloadPktSnip =
                  buildBootsrapConfigPacket();
                responsePayloadPktSnip =
                  gnrc_ipv6_esp_build(responsePayloadPktSnip, sa);
                gnrc_ipv6_esp_calc_icv(responsePayloadPktSnip, sa->sharedKey,
                                       sa->sharedKeyLength);
                send(responsePayloadPktSnip, &ip_addr);
            } else {
                LOG_ERROR("error on handle time sync request\n");
            }
            break;
        default:
            LOG_ERROR("PHASE: Unknown! Ignore package");
    }
}
```

Listing 4.10: C-Funktion zur Nachrichtenverarbeitung beim Sender

4.4.3 Paktenauthentifizierung

Befindet sich ein Client in der Phase des Datenempfangs *TESLA_DATA*, müssen wie das Kapitel 3.5 beschreibt, verschiedene Teilschritte ausgeführt werden um ein Paket mit TESLA zu authentifizieren. Wie diese Schritte im Prototypen umgesetzt worden sind, wird in den folgenden Abschnitten näher erläuert, wobei die Berechnung des aktuellen Intervals seperat betrachtet wird, da dies eine Kernfunktionalität darstellt. Anschließend wird auf das spezielle Vorgehen auf Sender- sowie auf Empfängerseite eingegangen.

Berechnung des aktuellen Intervals

Bevor Pakete mittels TESLA authentifiziert oder verifiziert werden können, ist es obligatorisch den aktuellen Interval I_i (Sender) bzw i_{kum} (Empfänger) zu ermitteln. Dies erfolgt mit der im Codeausschnitt 4.11 gezeigten Funktion *get_current_interval(start_time, clock_delta, duration_millis)*, die sowohl für Sender als auch Empfänger gleichermaßen genutzt werden kann und als Parameter die Startzeit T_0 *start_time*, die maximale Zeitabweichung T_{ME} *clock_delta* sowie die Länge des Intervalls T_{int} *duration_millis* in Millisekunden akzeptiert. Die Berechnung erfolgt entsprechend der Formel 3.9, wobei T_0 der Variable *time* entspricht und auf Senderseite $T_{ME} = 0$ ist. Abschließend wird auf die Ergebnisvariable *interval* eins aufaddiert, da das Ergebnis auch 0 sein kann, jedoch der erste Interval mit 1 indeziert ist.

```
1 uint32_t get_current_interval(uint32_t start_time, uint32_t clock_delta,
2                               uint32_t duration_millis) {
3     uint32_t time = start_time + clock_delta;
4     uint32_t now = time_now_millis();
5     uint32_t delta_time = now - time;
6     uint32_t interval = (delta_time / duration_millis) + 1;
7     return interval;
```

Listing 4.11: C-Funktion zur Berechnung des aktuellen Intervalls

Sender

Der Versand von mit TESLA authentifizierten Pakete ist zunächst analog zum Versand eines ESP Pakets und unterscheidet sich lediglich Anhand des ICVs. Da die Authentifizierung des ESP Headers nicht mit einem PSK oder einem durch IKE erhaltenen Schlüssel erfolgt, muss anhand des zuvor berechneten Intervals I_i ein Schlüssel k_i aus der Key-Chain selektiert werden. Mit diesem wird anschließend der ICV mittels der Funktion *gnrc_ipv6_esp_calc_icv(*esp,*key,key_size)* berechnet, wodruch ein valides ESP Paket entsteht. Zusätzlich müssen die für TESLA relevanten Authentifizierungsinformation an den ICV angehängt werden. Hierfür wird zunächst ein zweiter Schlüssel k_{pub} analog zur Formel 3.7 berechnet und in ein neu erzeugtes *Snip*, welches der im Codeausschnitt 4.12 abgebildeten C-Struktur entspricht, geschrieben. Zusätzlich wird das Feld *interval* auf den Wert des berechneten Intervals I_i gesetzt, um das Paket eindeutig einem Interval zuzuweisen.

Empfänger

Analog zum Versand eines Pakets muss beim Empfang eines mit TESLA authentifizierten Pakets, der aktuelle Interval i_{kum} ermittelt werden, um zunächst entsprechend der Formel

```c
typedef struct {
    uint32_t interval;
    unsigned char disclosedKeyData[KEY_LENGTH];
    uint16_t disclosedKeyLength;
} tesla_auth_header_t;
```

Listing 4.12: C-Struct Tesla Paket Header

3.12 feststellen zu können, ob das Paket innerhalb des im Paket angegebenen Interval i_p versand worden ist. Ist diese triviale Überprüfung erfolgreich, kann anschließend der im Paket enthaltene veröffentliche Schlüssel k_{pub}, falls vorhanden, nach einer ebenfalls erfolgreichen Verifizierung gespeichert werden. Schlägt die Verifizierung von k_{pub} fehl, wird das Paket verworfen und nicht weiter verarbeitet. Im positiven Fall wird das aktuelle Paket P_i zwischengespeichert und die Authentifizierung der bereits zwischengespeicherten Pakete, falls möglich, durchgeführt. Zur Authentifizierung der Pakete, wird über alle Pakete im Zwischenspeicher iteriert und geprüft ob für ein Paket P_i ein veröffentlichter Schlüssel k_{i_p} vorhanden ist bzw berechnet werden kann. Ist dies der Fall, so kann mit der Funktion *gnrc_ipv6_esp_validate_icv(*esp, *key, *key_size)* das Paket validiert werden und gibt im Erfolgsfall 0 zurück. Anschließend kann auch der ESP Header validiert und somit auf Modifikation überprüft werden, in dem die SPI sowie die Sequenznummer mit den Erwartungswerten aus der SA verglichen werden. Sind all diese Überprüfungen für ein Paket P_i erfolgreich so kann es aus dem Zwischenspeicher entfernt und zur Weiterverarbeitung an die Funktion *teslaHandlePayloadFunc(*payload)* übergeben werden, andernfalls wird das Paket ebenfalls aus dem Zwischenspeicher entfernt und verworfen.

```c
if (gnrc_ipv6_esp_validate_icv(esp, validationKey,
                               clientRunConfig.disclosed_key_len) == 0) {
    ipv6_ext_esp_header_hdr_t *esp_hdr = esp->data;
    if (gnrc_ipv6_esp_validate_header(esp_hdr, sa) == 0) {
        gnrc_pktsnip_t *payload = handle_pkt_ip_esp_payload(esp);
        if (teslaHandlePayloadFunc == NULL) {
            LOG_ERROR("handle payload function not set\n");
        } else {
            teslaHandlePayloadFunc(payload);
        }
    } else {
        LOG_ERROR("Error on esp header validation. Package discarded\n");
    }
} else {
    LOG_ERROR("Tesla package authentication failed. Package discarded\n");
}
```

Listing 4.13: Tesla Paketauthentifizierung

4.5 Programmstruktur

Nach dem in den vorangegangenen Abschnitten die Implementierung des Prototypen erklärt wurden, soll abschließend die Struktur sowie die Abhängigkeiten zwischen den Modulen des Prototyps selbst, sowie zu denen, welche von Riot-OS bereitgestellt werden, betrachtet wer-

den. Dafür illustriert die Abbildung 4.5 eine primitive Übersicht der verwendeten Module und soll zugleich die Zuständigkeiten der einzelnen Module innerhalb des Prototypen aufzeigen. Somit sind sämtlichen Funktionalitäten bezüglich Key-Chain in dem Module *crypt* untergebracht und wird ausschließlich vom Kernmodul *tesla*, welches die gesammte Logik zum Protokoll TESLA enthält, referenziert. Die Umsetzung von ESP, die im Abschnitt 4.2 beschrieben ist, ist im Modul *ipv6_ext_esp* enthalten und referenziert außerdem das *networking* Modul, in welchem alle grundlegenden Funktionen des GNRC Stacks betreffend angesiedelt sind. Die Abhängigkeiten zu Riot-OS sind neben dem obligatorischen Kernel überschaubar und begrenzen sich auf das Modul *hashes*, welches von Modul *crypt* benötigt wird, sowie den Modulen *gnrc_ipv6* und *gnrc_ipv6_ext*, die wiederum von den Modulen *ipv6_ext_esp* und *networking* referenziert werden.

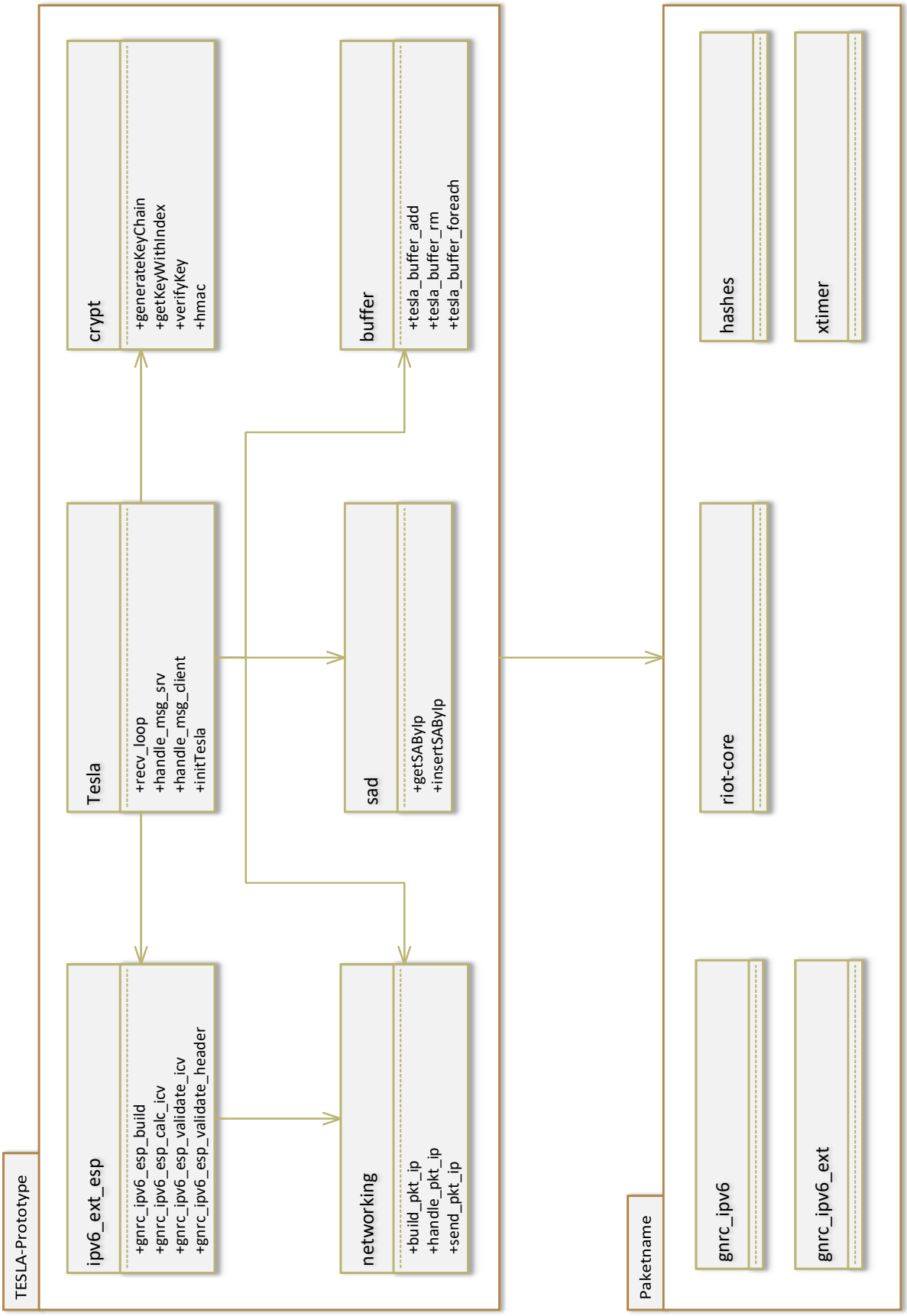

Abbildung 4.5: Codestruktur des Prototypen

5 Testbed

Zur Evaluierung der im Kapitel 4 beschriebenen Implementierung des Prototyps, muss eine Testumgebung sowie das Szenario spezifiziert werden. Wie zu Beginn dieser Arbeit eingeleitet wurde, soll das Szenario eines Lichtschalters in einem Smart Home näher betrachtet werden um eine Aussage darüber treffen zu können ob die Verwendung von TESLA in einem solchen Umfeld praktikabel ist. Wie ein möglicher Anwednungsfall aussehen kann, zeigt die Abbildung 5.1, wobei ein intelligenter Lichtschalter alle ebenfall intelligenten sich im Raum befindlichen Lampen und Leuchten steuern können soll. Daraus folgt, dass der Lichtschalter die Rolle des Senders annimmt und auf Grund der günstigeren Platzverhältnisse, Raum für ein Gerät bietet welches weniger Ressourceneinschränkungen aufweist als die Systeme in Lampen und Leuchten. Des weiteren sind ingesammt fünf Lampen und Leuchten im Raum platziert, die als Empfänger fungieren und in unterschiedlichen Abständen zum Lichtschalter (Sender) positioniert sind.

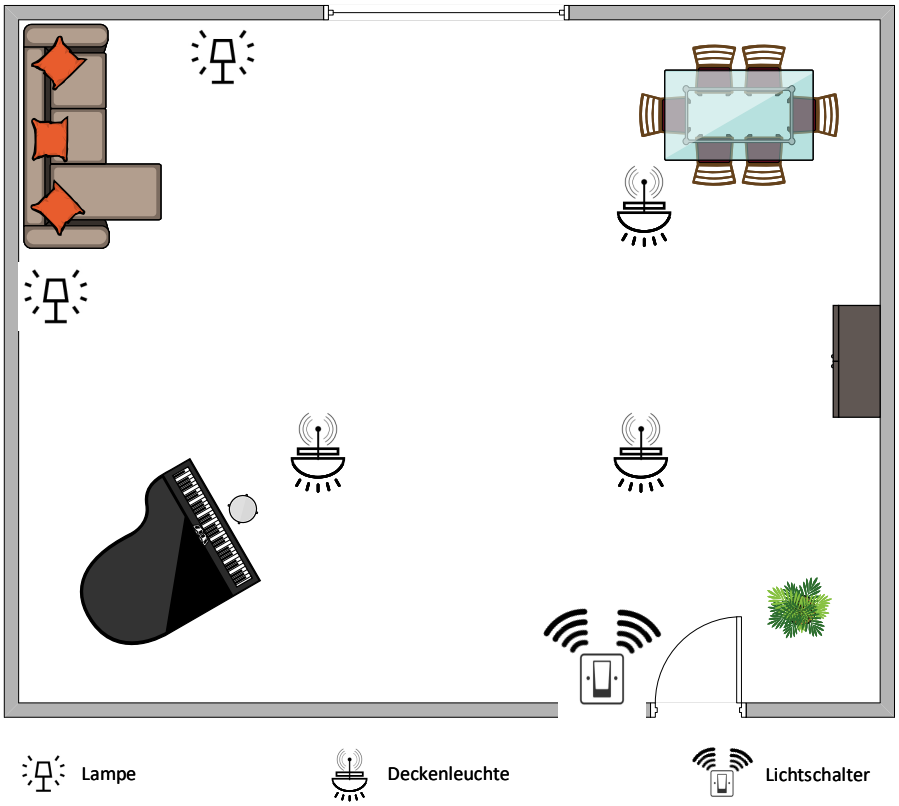

Abbildung 5.1: Beispiel für ein Smart Home Szenario

5.1 Testumgebung

Als Umgebung zum Testen der Implementierung wurde das IoT-Lab [10] gewählt. Das IoT-Lab ist eine wissenschaftliche Testumgebung und stellt fast 1800 Geräte verschiedener Typen zur Verfügung. Somit ist es Wissenschaftlern möglich schnell und einfach ein Netzwerk aus verschieden Geräten aufzubauen und Messwerte zu sammeln, in dem voller Zugriff auf die Gateways, zu denen die Geräte verbunden sind, gewährt wird.
Für die folgenden Tests wurden Knoten des IoT-Lab-Netzwerks gewählt, welche auf einem ARM Cortex M3 Mikrocontroller basierend und im Folgenden als M3-Knoten bezeichnet werden. Die M3-Knoten besitzen eine CPU mit einem Takt von maximal 72 MHz, 64 KB RAM und zwischen 256 und 512 KB ROM. Somit lässt sich der M3-Knoten zu den Geräte der Klasse 2 zuordnen und zählt außerdem zu den von Riot-OS unterstützten Geräten.

5.2 Testszenarien

Zum Testen des Protokolls TESLA mit dem im Kaptiel 4 beschriebenen Prototypen muss analog zum Design des zuvor beschriebenem Smart Home Szenario, ein Experiment im IoT-Lab erstellt werden. Dabei soll zunächst betrachtet werden ob eine Anwendung in einem nachgestellten Smart Home Szenario prinzipiell möglich ist oder nicht.
Um verschiedene Szenarien im IoT-Lab abzubilden, bietet die Web-Oberfläche zum Konfigurieren eines Experiments eine Übersicht (siehe Abbildung 5.2) sämtlicher verfügbaren Geräte im IoT-Lab. Diese Übersicht, in welcher jeder Punkt ein Gerät repräsentiert, erleichtert die

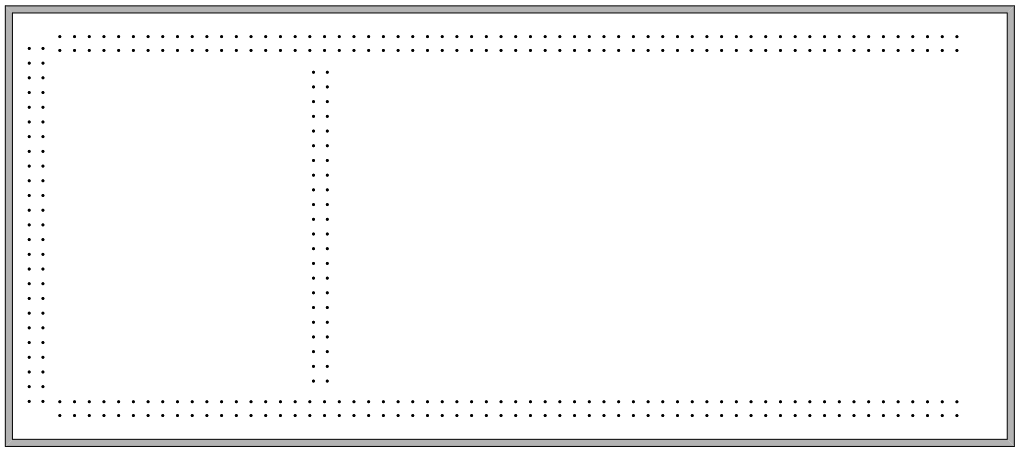

Abbildung 5.2: Geräteübersicht im IoT-Lab

Auswahl bestimmter M3-Knoten, welche für ein Experiment verwendet werden sollen. Des Weiteren beschreibt jeder Punkt die Position des jeweiligen Gerätes am IoT-Lab Standort, wodurch der Aufbau des Experiments zum Nachstellen des in Abbildung 5.1 gezeigten Smart Home Szenarios vereinfacht wird.

5.2.1 Ein Sender ↔ ein Empfänger

Das erste und einfachste Testszenario begrenzt sich zunächst auf zwei M3-Knoten, wobei einer die Rolle des Senders und der andere die des Empfängers übernimmt. Wie in Abbildung 5.3 gezeigt wird, liegen zwischen den zwei gewählten M3-Knoten (größere Punkte) vier weitere unbenutzte Knoten, was für diese und alle weiteren Experimente als Entfernungsmaß dient. Weiterhin repräsentiert der schwarze Punkt (●) den Sender und der graue Punkt (●) den Empfänger. Ziel dieses Experiments mit einem trivialen Aufbauen soll sein, eine Aussage darüber treffen zu können ob ein Datenaustausch bei einem Abstand von vier Knoten zwischen den beiden Endpunkte möglich ist.

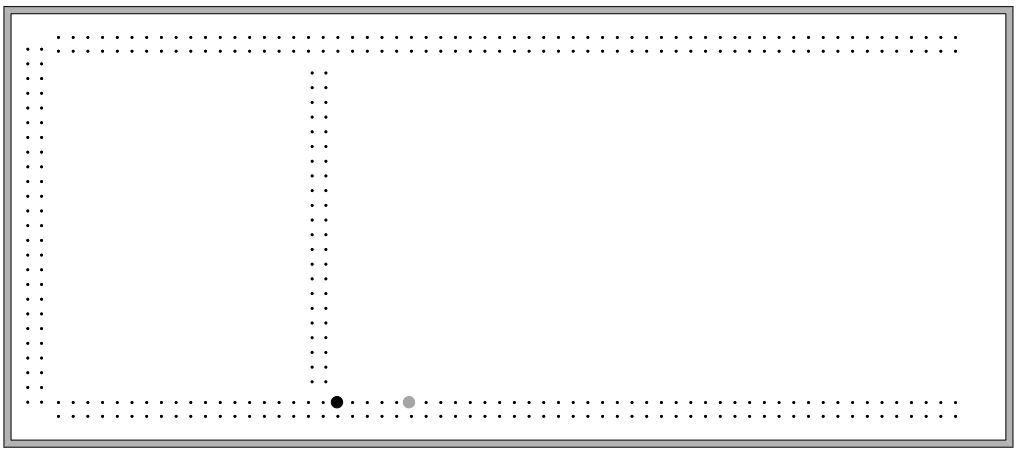

Abbildung 5.3: Aufbau des Experiments im IoT-Lab mit einem Sender (●) und einem Empfänger (●)

5.2.2 Ein Sender ↔ fünf Empfänger

Der Aufbau des zweiten Experiments umfasst ingesammt sechs Knoten, wobei ein Knoten die Rolle des Senders und die restlichen fünf Knoten die des Empfängers übernehmen. Dieses Experiment soll das in Abbildung 5.1 gezeigte Szenario möglichst idententisch nachbilden, wobei ähnlich zum ersten Experiment ein maximaler Abstand von vier Knoten gewählt wurde, wie die Abbildung 5.4 zeigt. Unter der Annahme, dass das erste Experiment erfolgreich war und eine Kommunikation zwischen den Endpunkte möglich ist, liegt der Fokus dieses Experiments darauf zu prüfen, bis zu welchem Abstand eine Kommunikation ohne Router möglich ist. Außerdem wird die Zeitverzögerung T_δ in Millisekunden betrachtet, um möglicherweise eine obere Grenze von T_δ zu finden und feststellen zu können ob T_δ eine Aussagekraft darüber besitzt, ob eine sichere Kommunikation mit TESLA möglich ist.

5.2.3 Ein Sender ↔ viele Empfänger

In einem weiteren Experiment soll untersucht werden, wie sich TESLA mit mehreren Empfängern verhält. Wobei die Anzahl der Empfänger des vorangegangenen Experiments mit fünf Empfängern überschritten werden soll. Desweiteren wird kein Mindestabstand von vier Knoten, wie es in

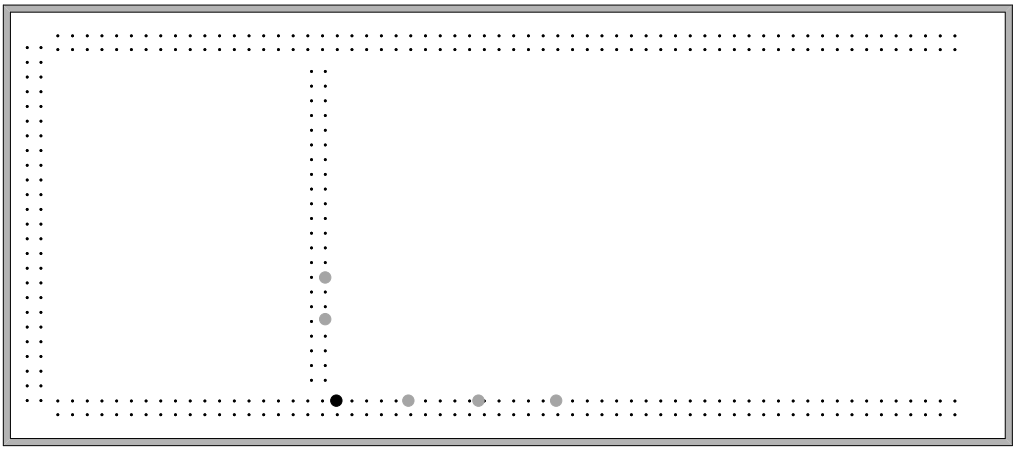

Abbildung 5.4: Aufbau des Experiments im IoT-Lab mit einem Sender (●) und fünf Empfängern (●

den ersten beiden Experimenten der Fall ist, eingehalten um die Anzahl der Geräte im Senderadius des Senders zu maximieren. Dabei soll zunächst geprüft werden ob es möglich ist, alle Empfänger gleichzeitig zu initialisieren, wodurch sich eine Aussage darüber treffen lässt wie robust das Protokoll sowie die Implementierung ist. Ist die Grenze von initialisierten Empfängern, welche in diesem Experiment auf 10 gesetzt ist, erreicht oder überschritten, sollen mehrere mit TESLA authentifizierte Nachrichten versandt werden. Zu erwarten ist dabei, dass alle Empfänger mindestens eine Nachricht erfolgreich authentifizieren können, sobald mindestens zwei Pakete empfangen wurden und die Differenz der Intervallindizes größer oder gleich d sind. Da ein begrenzter Senderadius eines M3-Knoten zu erwarten ist, werden die Geräte, wie die Abbildung 5.5 illustiert, in einem Block positioniert.

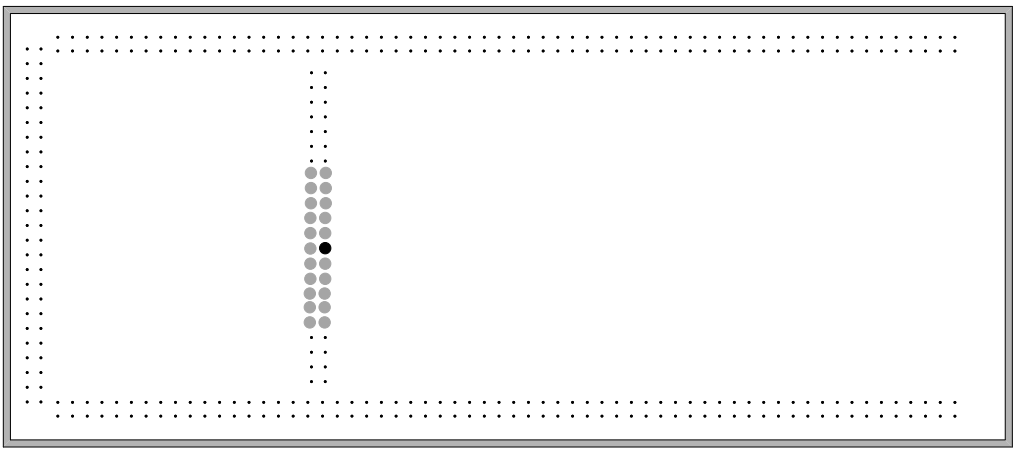

Abbildung 5.5: Aufbau des Experiments im IoT-Lab mit einem Sender (●) und viele Empfänger (●)

5.2.4 Maximaler Abstand zwischen Sender und Empfänger

Mit Hilfe dieses Experiments soll überprüft werden, ob die Kommunikation zwischen zwei Knoten möglich ist, die mit einem maximal möglichen Abstand (begrenzt durch das IoT-Lab) positioniert sind. Da die M3-Knoten bedingt durch die verfügbaren Ressourcen nicht über ausreichend Sendereichweite verfügen, werden zusätzliche Knoten die als Router aggieren im Experiment eingeschlossen. Dabei sind die alle Knoten zwischen Sender und Empfänger Router, da sie weder explizit senden noch ausreichend initialisiert sind um als TESLA Empfänger zu gelten. Somit ist die Kommunikation zwischen den beiden Endpunkte theoretisch gewährleistet. Um unabhängig des Senderadius eines M3-Knoten zu sein, werden analog zum Experiment im Abschnitt 5.2.3 alle Knoten mit einem möglichst geringen Abstand zu einander positioniert. Ziel des Experiments soll zeigen oder wiederlegen, dass TESLA mit einer hohen Zeitverzögerung T_δ der Pakete funktionieren kann.

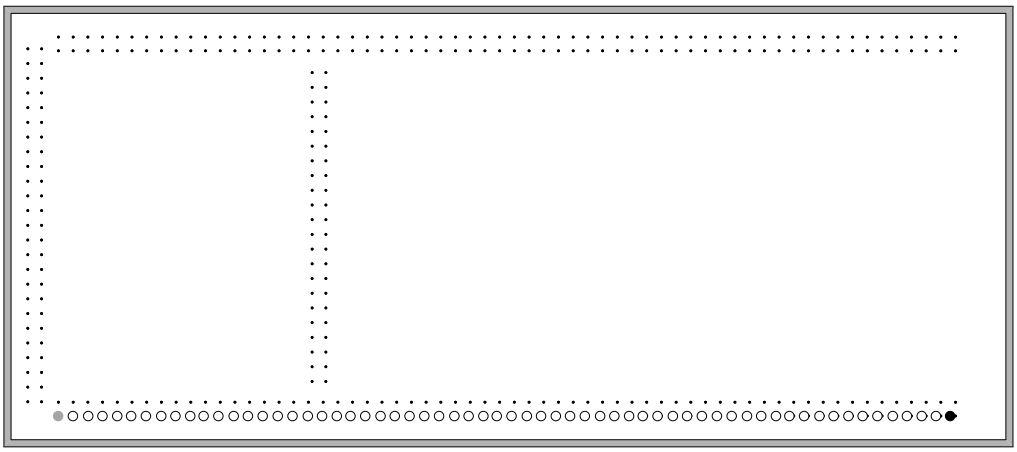

Abbildung 5.6: Aufbau des Experiments im IoT-Lab mit einem Sender (●), einem Empfänger (●) und mehreren Routern

6 Verwandte Arbeiten

Authentifizierung von Multicast Datenpaketen ist ein fortwährendes Problem, wobei das in dieser Arbeit beschriebene und implementierte Protokoll TESLA nicht die einzige existierende Lösung darstellt. In den folgenden Abschnitten sollen mögliche alternative Lösungsansätze zu TESLA sowie Abwandlungen vorgestellt werden. Da Speicher nicht ausschließlich bei der Größe der übertragenen Datenpakete von Relevanz ist, wird zusätzlich eine leichtgewichtige Variante eines X.509 Zertifkats betrachtet.

6.1 EMSS

In den Artikeln *Efficient authentication and signing of multicast streams over lossy channels* [27] sowie *Efficient multicast packet authentication using signature amortization* [26] wird von den Autoren neben TESLA auch das alternative Protokoll Efficient Multi-chained Stream Signature (EMSS) vorgestellt. EMSS ist unter Anderem für Umgebungen konzipiert in denen Zeitsychronisation nicht möglich oder nur sehr schwer umsetztbar ist. Dabei wird eine Kombination von HMAC und digitalen Signaturen verwendet, wobei im Vordergrund steht, dass der Versandt eines Pakets nicht abstreitbar ist. Anders als bei TESLA sind bei EMSS nicht die verwendeten Schlüssel zur Berechnung der HMAC miteinander verknüpft sondern die Pakete selbst. Außerdem wird anstelle eines Schlüssels, welcher der Authentifizierung der Pakete dient, ein Signaturpaket zeitversetzt versand, wodurch ähnlich zu TESLA eine zeitverzögerte Verifikation der Daten auf Empfängerseite erfolgenen kann. Die grundle-

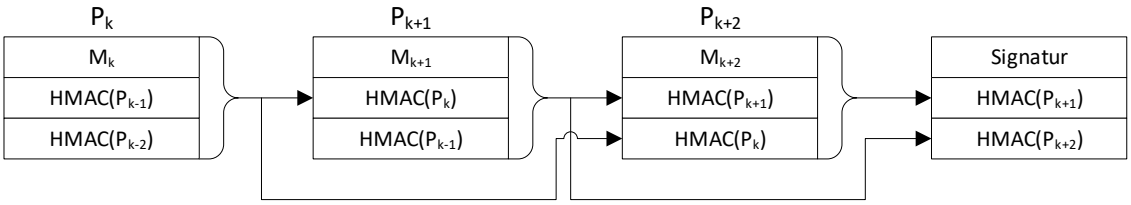

Abbildung 6.1: EMSS Beispiel zur Paket-Abhängigkeit

gende Idee von EMSS ist, dass der HMAC von Paket P_k mit den Daten des Pakets P_{k+1} und der HMAC von P_{k+1} mit den Daten des Pakets P_{k+2} konkatiniert wird. Anschließend an die drei Pakete folgt ein Paket mit einer digitalen Signatur über den $\text{HMAC}_{P_{k+2}}$ des letzten Pakets P_{k+2}, wodurch die Pakete miteinander verknüpft sind und der Versand nicht mehr abstreitbar ist. Im Falle von Paketverlust, müssen jedoch mehrere Pakete verworfen werden, weshalb EMSS dieses Vorgehen erweitert und die Robustheit gegen Paketverlust erhöht. Wie in der Abbildung 6.1 illustiert, enthält ein EMSS Paket P_k die HMACs von P_{k-1} und P_{k-2}, sowie das Paket P_{k+1} die HMACs von P_k und P_{k-1} usw. Analog zur Grundidee wird anschließend ein Paket mit einer Signatur verschickt. Um die Tolleranz gegen Paketverlust

weiter zu erhöhen, ist es möglich das Paket, welches die Signatur beinhaltet mehr als einmal zu verschicken. Dieses Verfahren reduziert die Verzögerung bei der Verifikation der Daten sowie den benötigten Speicherbedarf zum Zwischenspeichern der Paket auf Empfängerseite.

6.2 μTESLA

Die Erkenntnisse aus dem Kapitel 7.2 zeigen, das TESLA nicht für Geräte der Klasse 0 und 1 konzipiert ist. Wie Perrit at AL. in ihrem Paper *SPINS: Security protocols for sensor networks* [30] beschreibt, senden Geräte in Sensornetzen Pakete mit einer durchschnittlichen Größe von 30 Bytes, TESLA wiederum hat aproximativ einen Overhead von 24 Bytes pro Paket und die in dieser Arbeit beschriebenen Implementierung zusätzlichen Overhead durch die Protokolle IPv6 und ESP, was die Verwendung auf Geräten der Klasse 0 und 1 unmöglich macht. Außerdem ist es, auf Grund der geringen Speicherkapazitäten, nicht möglich eine Key-Chain in den Speicher eines Sensorknotens abzulegen. μTESLA wurde für den Einsatz in Sensornetzen entwickelt und löst den asymetrischen kryptografischen Mechanismus durch einen symmetrischen Mechanismus ab. Folglich wird das Veröffentlichen von Schlüsselmaterial in jedem Paket zur Verifikation der Daten obsolet und reduziert somit den Overhead um die Länge des mitgesendeten Schlüssels. Da das Vorgehen zur Verifikation der Datenpakete nicht vom ursprünglichen TESLA abweicht müssen dennoch Schlüssel verteilt werden. Dies erfolgt in dem periodisch Pakete mit einem symmetrischen Schlüssel publiziert werden, welcher analog zu TESLA einem Zeitinterval i zugeordnet ist. Weiter benötigten symmetrische Verfahren signifikant geringeren Rechenaufwand und sind dadurch besser für Geräte mit stark limitierer Rechenleistung geeignet. Ähnlich zu EMSS kann die Tollerenz gegenüber von Paketverlust dahingehend reduziert werden indem die Anzahl der versendeten Schlüsselpakete pro Interval erhöht werden.

6.3 infTESLA

Wie im Abschnitt 3.2 ist jedem Zeitintervall i ein kryptografischer Schlüssel k_i aus der Key-Chain der Länge N zugewiesen. In der ursprünglichen Spezifikation zu TESLA ist jedoch keine Vorgehensweise beschrieben wie ein Überlauf zu behandeln ist, was im Umkehrschluss bedeutet dass alle Empfänger erneut initialisiert werden müssen sobald der letzte Schlüssel aus der Key-Chain verwendet wurde. Das in dem Paper *Multicast Delayed Authentication For Streaming Synchrophasor Data in the Smart Grid* [4] beschriebene Protokoll *infinity*-TESLA (infTESLA) ist eine Modifikation von TESLA und erweitert die ursprüngliche Spezifikation dahingehend, dass keine erneute Senderinitialisierung erforderlich ist. infTESLA versucht das Problem zu addressieren in dem es *Dual Offset Key Chains* implementiert. Hierfür wird eine zusätzliche Key-Chain auf Senderseite erzeugt und verwaltet, die Paketstruktur erweitert, sowie die Initialierungsphase und die Paketauthentifizierung angepasst. Bedingt durch die zusätzliche Key-Chain muss bei der Empfängerinstalisierung ein weiterer Schlüssel zu Verifikation der folgenden Schlüssel sowie ein weiterer Index für diesen übertragen werden. Die ursprüngliche TESLA Paketstruktur wird analog zu den Paketen der Intialisierung um zwei weitere Felder erweitert wodurch das zu sendene Paket neben den Nutzdaten M, zwei Intervalindizes i und j, zwei veröffentliche Schlüssel k_{pub_1} und k_{pub_2} und ebenfalls ein HMAC k'_{ij} beinhaltet. Die Berechnung des HMAC erfolgt grundlegend idententisch zur ursprünglichen Definition mit der Abweichung, dass im Vergleich zu TESLA die

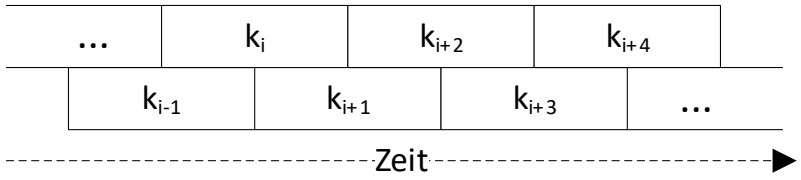

Abbildung 6.2: infTESLA Dual Offset Key-Chain

Schlüssel k_i und k_j konkatiniert als Eingabeparameter für die Hash-Funktion F' fungieren. (siehe zum Vergleich die Definition 3.8)

$$P_M = \{M||i||j||k'_{ij}||k_{pub_1}||k_{pub_2}\}$$

Dabei stellen k_i und k_j sowie auch k_{pub_1} und k_{pub_2} je einen Schlüssel aus beiden Key-Chains dar. Weil das beim Senden zwei Schlüssel zur Verifikation vorliegen ist es somit möglich auf Senderseiter eine neue Key-Chain zu erzeugen ohne das die Empfänger erneut initialisiert werden müssen. Der Sender publiziert einen Schlüssel aus der neu erzeugten Key-Chain und berechnen einen HMAC mittels einem Schlüssel aus der zweiten (noch nicht übergelaufenen) Key-Chain wodurch die Empfänger die Authentizität des neuen Schlüsselmaterials überprüfen können. Folglich scheint infTESLA das Problem von auslaufenden Key-Chains und das Vermeiden von Reinitialisierungen zu lösen. Ein weiterer Gewinn soll laut den Authoren Câmara, Anand, Pillitteri und Carmo in der Performance liegen, wobei annähernd 16 Prozent weniger Rechenzeit auf der Senderseite und sogar etwas über 47 Prozent weniger Rechenzeit auf Empfängerseite benötigt werden soll. Bedingt durch die weitere Key-Chain und die zusätzlichen Felder für den zweiten Schlüssel kann der zusätzlich benötigte Speicherplatz bei Sender sowie Empfänger als nachteilig betrachtet werden.

6.4 TESLA++

Ein Denial-of-Service (DoS) Angriff ist eine Variante zu versuchen einen Endpunkte zu überlasten bis dieser nicht mehr auf Anfragen anderer Netzwerkentitäten antworteten kann. Dabei sendet ein Angreifer möglichst viele Packete, welche beim Angriffsziel zunächst verarbeitet werden müssen bevor entschieden wird ob diese verworfen werden können oder nicht. Moderne Computersysteme verfügen heutzutage allerdings über ausreichend Kapazitäten um erreichbar zu bleiben. Ist das Angriffsziel jedoch ein Geräte mit beschränkten Ressourcen, verfügt es sehr wahrscheinlich nicht über ausreichend Ressourcen um dem Angriff Stand zu halten. TESLA ist nach Spezifikation bereits gut gegen solche Angriffe geschützt, wobei eine simple DoS Attacke mehr auf eine Überlastung der Rechenkapazitäten ausgelegt ist, zielt eine andere Variante des DoS Angriffs auf die Ausreizung der Speicherkapazitäten. Da TESLA sämliche Pakete zunächst zwischenspeichert werden, wird bei einem derartigem Angriff versucht möglichst viel Speicher des Gerätes zu belegen bis diesem nicht mehr ausreichend freier Speicherplatz zur Verfügung steht und folglich eingehende Pakete nicht mehr verarbeitet bzw. zwischenspeichert werden können. TESLA++ [32] ist eine weitere Abwandlung von dem ursprünglichen Protokoll TESLA. Es bietet besseren Schutz vor speicherorientierten DoS Angriffen und offeriert zugleich eine geringere Ressourcenbeanspruchung auf der

Empfängerseite.
Diese Verbesserung werden erreicht indem der Hashwert k_i' zu den korrespondieren Daten nicht im gleichen Paket versandt wird, sondern in einem dediziertem Paket inklusive dem zugehörigen Interval bzw. Schlüsselindex i enthalten ist. Zusätzlich erfolg der Versand von $P_{HMAC} = \{k_i'||i\}$ vor dem Versand der eigentlichen Daten. Durch den dedizierten Versandt von P_{HMAC} verringert sich die Größe des Datenpakets P_m und enthält lediglich die Nutzdaten M, den Schlüssel k_i zum Berechnen von k_i' sowie den zugehörigen Index i, wie in Abbildung 6.3 beispielhaft illustiert. Dieses Vorgehen erhöht die Tolleranz gegenüber DoS Angriffen, da dies zum Einen die Angriffskomplexität erhöht und zum Anderen der Empfänger in der Lage ist ein Paket zu welchem kein Hashwert vorliegt sofort zu verwerfen anstatt dieses zunächst zwischenspeichern zu müssen. Eine weitere Anpassung, bedingt durch die Sendereihenfolge der Pakete, erfolgte auf Empfängerseite. Der Empfänger speichert nicht wie in TESLA die Pakete zwischen sondern die Hashwerte, wodurch ein geringerer Speicherbedarf beim ihm notwendig ist. Zusätzlich speichert dieser nicht den empfangenen Hashwert und den Index sondern berechnet einen neuen Hashwert $HMAC_R$ mit k_i' und einem arkanen nur dem Empfänger bekannten Schlüssel k_R als Eingabeparameter. Der $HMAC_R$ ist nochmals kleiner und den Speicherverbrauch für die Zwischenspeicherung erneut reduziert. Analog zu TESLA wird der Hashwert zusammen mit dem empfangenen Index zwischengespeichert bis das korrespondierende Paket beim Empfänger eingegangen ist. Dabei unterscheiden sich nur die zwischengespeicherten Daten zu dem in TESLA spezifiziertem Verhalten. Sobald ein Datenpaket P_m beim Empfänger eingegangen ist, kann dieses ohne Verzögerung authentifiziert werden in dem die Hashwerte berechnet und im Zwischenspeicher gesucht werden. Ist kein Hashwert $HMAC_R$ zu diesem Paket zu finden, kann es verworfen werden. Zur Suche im Puffer muss zunächst der Hashwert $HMAC_{R_P}$ zu dem eingegangen Paket berechnet werden. Dazu muss der ursprüngliche Hashwert k_{i_p}' mit

$$F(M, k_i) = k_i'$$

ermittelt werden um anschließend $HMAC_{R_P}$ berechnen zu können. Die Hashfunktion F ist die gleiche wie sie vom Sender verwendet wird und nimmt die Nutzdaten M und den dazugehörigen Schlüssel k_i an.

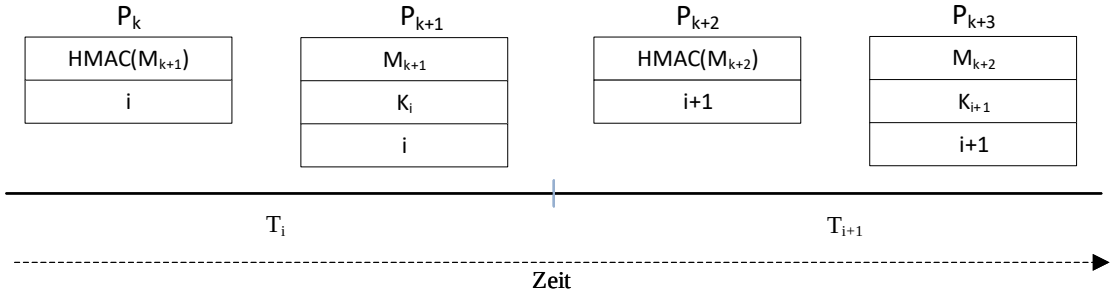

Abbildung 6.3: Beispiel: Paketreihenfolge mit TESLA++

6.5 Lightweigth X.509 Certificate

Neben der in dieser Arbeit für den Prototypen gewählten Variante einen PSK zur Authentifizierung der Kommunikationspartnern zu verwenden, existiert die Möglichkeit sich mit einem X.509 Zertifikat [6] gegenseitig zu authentifizieren. Diese Art von Zertifikaten sind allerdings für herkömmliche Computer oder Server entwickelt worden und nicht darauf ausgelegt auf Geräten mit eingeschränkten Ressourcen zu funktionieren. Aus diesem Grund haben die Authoren *Forsby, Filip and Furuhed, Martin and Papadimitratos, Panos and Raza und Shahid* eine mögliche Lösung erarbeitet und sie in dem Paper *Lightweight X. 509 Digital Certificates for the Internet of Things* [11] veröffentlicht. Darin beschreiben sie wie sich ein konvetionelles X.509 Zertifikat komprimieren lässt um es komptibel zu Geräte mit beschränkten Ressourcen zu machen und gleichzeitig nicht vom Standard abweicht. Durch die Kompression einzelner Felder im Zertifkat ist es möglich die Größe auf ein Drittel der ursprünglichen Größe zu reduzieren und damit nur noch knapp 150 Bytes Speicher zu belegen. Durch die Komformität zum Standard ist es außerdem möglich die leichtgewichtige Variante eines X.509 Zertifikates mit jeder beliebigen Public Key Infrastructure (PKI) zu nutzen.

6.6 TESLA on Linux

Zum Zeitpunkt des Entstehens dieser Arbeit arbeitet der Student Jonas Dellinger in seiner Arbeiter *Streaming Multicast Authentication with TESLA and ESP on Linux* [7] ebenfalls daran, Netzwerkverkehr in Multicast-Szenarien mit Hilfe von TESLA und ESP zu authentifizieren. Als Basis für die Implementierung des Protypten wird eine Linux-Distribution gewählt und ebenfalls mit der Programmiersprache C umgesetzt. Ziel seiner Arbeit ist es ein Aussage darüber treffen zu können ob und wie man TESLA in den ESP Netzwerkstack von Linux einbetten kann. Dazu stellt er ein Konzept vor wie der ESP-Header für TESLA adaptiert wird, um die Bedeutung der Felder nicht zu verändern sondern vielmehr um die Funktionalität von TESLA zu erweitern. Zusätzlich wird der Speicherverbrauch, CPU-Zeit und die Auslastung des Netzwerks analysiert und abschließend die protoypische Implementierung bekannten Authentifizierungsprotokollen gegenüber gestellt.

7 Evaluation

Bevor die anfangs gestellte Frage ob das Protokoll TESLA in einem Smart Home Szenario funktionieren kann beantwortet wird, soll zunächst die Performance von Key-Chains und anschließend der benötigte Speicherbedarf betrachtet werden. Hier für wurde die Zeit zur Erzeugung einer Key-Chain sowie die Zugriffszeiten im Mittel auf einem ARM Cortex M3 Gerät gemessen um eine Aussage darüber treffen zu können welche Key-Chain spezifischen Konfigurationsparameter optimal sind um das Verhältnis zwischen Speicherbedarf und Laufzeit bei Schlüsselselektionen möglichst ausgewogen zu halten. Anschließend daran wird im Abschnitt 7.2 der Speicherbedarf der prototypischen Implementierung von TESLA mit ESP analysiert um zu bestimmen, welche Klasse an Geräte mindestens zu verwenden sind damit das Protokoll mit einer Applikation verwenden werden kann.

7.1 Hash-Chain-Performance

Zur Evaluierung der Performance der im Rahmen dieser Arbeit implementierten Key-Chain wurden 3 Kennzahlen definiert, die gemessen bzw berechnet werden können.

1. Zeit die benötigt wird um eine Key-Chain initial zu Erzeugen.
2. Speicherverbrauch der erzeugten Key-Chain inklusive Wegpunkte
3. Zeit die benötigt wird um einen Schlüssel in der Key-Chain zu selektieren.

Die Messdaten zu den folgenden Diagrammen wurden zum gleichen Zeitpunkt auf fünf verschiedenen M3-Knoten erfasst und jeweils auf diesen fünfmal wiederholt. Daraus ergeben sich 25 Messungen für jede der folgenden Konfigurationen.

Schlüssellänge 256 Bit (32 Byte)

Schlüsselanzahl 1024

Wegpunkte 0,2,4,8,16,32,64,128,256,512

Um eine valide Aussage über die unter 3.2 aufgezeigten Speicheroptionen treffen zu können, wurde für die Zeit bei der Erstellung der Key-Chain mit unterschiedlicher Anzahl an Wegpunkten gemessen. Das Diagram in der Abbildung 7.1 illustiert die gemessenen Zeiten für eine Key-Chain mit 1024 Schlüsseln und der auf X-Achse angegebenen Wegpunkten. Außerdem ist abzulesen, dass die benötigte Zeit mit mehr als 64 Wegpunkte signifikat ansteigt und ein Zuwachs von fast einer Millisekunde zu verzeichnen ist. Zusätzlich überstreiten die gemessenen Zeiten die durchschnittliche Zeit bei einer Key-Chain mit ca. 100 Wegpunkten und mehr. Dies lässt jedoch noch keine valide und vergleichbare Aussage über die Performance der Implementierung zu. In dem Diagram in Abbildung 7.2 wird sichtbar, dass im Mittel die Zugriffszeiten mit der Anzahl der Wegpunkte in der Key-Chain korrelieren. Mit

steigender Anzahl an Wegpunkten fällt die Zeit zur Schlüsselselektion signifikant ab, wobei sich schlussfolgern lässt, dass bei Verdopplung der Wegpunkte sich die Zugriffszeit im Mittel halbiert und den gemeinsamen Mittelwert bereits ab 8 Wegpunkten deutlich unterschreitet. Parallel zu der ansteigenden Anzahl an Wegpunkten steigt der benötigte Speicherbedarf der

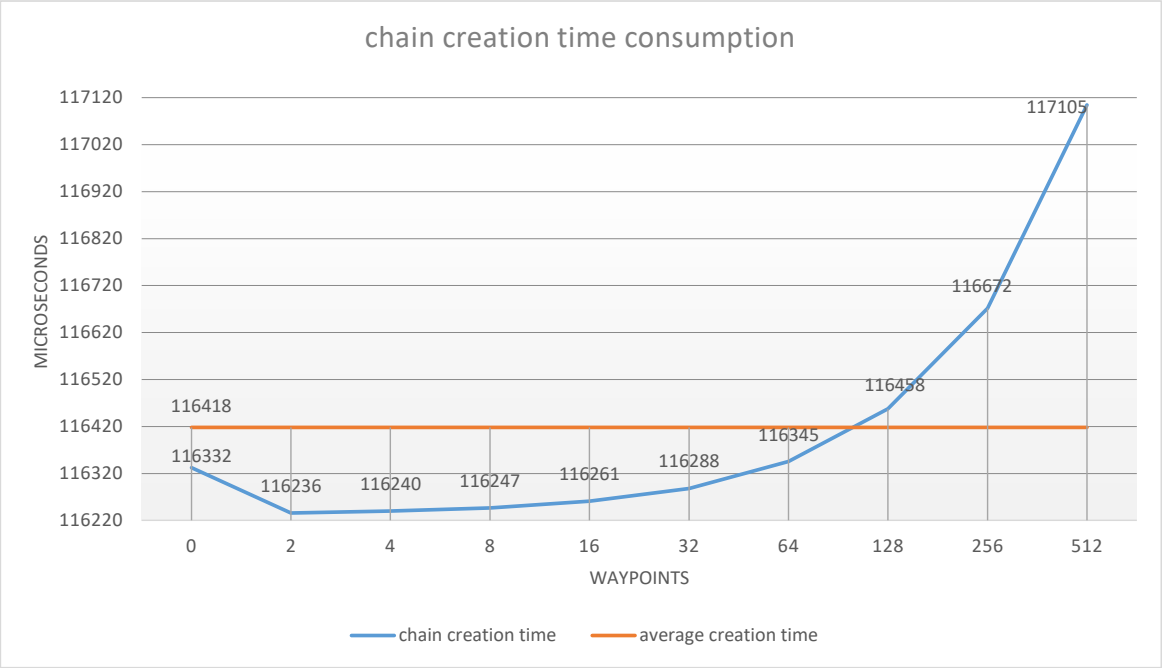

Abbildung 7.1: Mittlere Zeit in Microsekunden zur Erstellung der Key-Chain mit unterschiedlicher Anzahl an Wegpunkten

Key-Chain und umfasst bereits ein Kilobyte (1 KB = 1024 Byte) (KB) für 32 Wegpunkte inklusive des initialen Byteblock (Seed) und dem letzten Schlüssel. Die Konvergenz der Kurve für die Zugriffszeiten lässt sich mit den Einzelmessung in Abbildung 7.3 weiter erhärtertet. In diesem Diagramm ist die Zeit zur Selektion von Schlüsseln zwischen zwei Wegpunkten deutlich in der steigenden Flanke abzulesen. Die Ursache des Sägezahn-ähnlichem Muster liegt in der Implementierung. So kann ein vorhandener Schlüssel mit $O(1)$ selektiert werden wobei Schlüssel zwischen den Wegpunkten zunächst berechnet werden müssen und folglich mit $O(N/N_{wp})$ selektiert werden können. Dabei gibt N die Länger der Key-Chain und N_{wp} die Anzahl von Wegpunkten an. Betrachtet man in Abbildung 7.3 ebenfalls die Werte unter dem Durchschnitt lässt sich eine valide Aussage zu den annähernd optimalen Konfigurationsparameter treffen.
Für ein ausgeglichenes Verhältnis zwischen Zugriffszeit und Speicherverbrauch lässt sich für eine Key-Chain der Länge 1024 die Wegpunkteanzahl von 32 bei einem Speicherbedarf von ca. einem KB definieren. Das hat zur Folge, dass ein Schlüssel zwischen zwei und fünf Millisekunden berechnet werden kann und zur weiteren Verwendung zur Verfügung steht.
Der Zeitkorridor zur Schlüsselselektion bleibt auch für längere Key-Chains mit gleichbleibenden Abstand zwischen den Wegpunkten unverändert und beeinflusst lediglich den Speicherbedarf, wodurch zum Beispiel für Key-Chains der Länge 2048 zwei KB reserviert werden müssen.

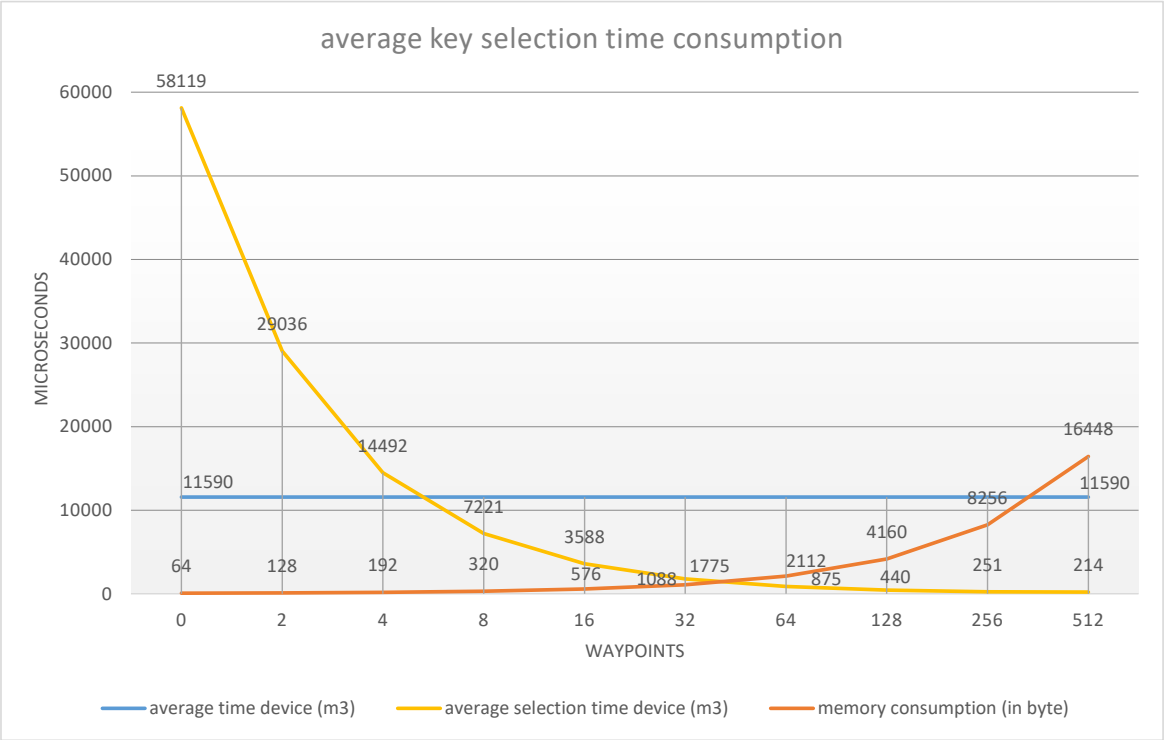

Abbildung 7.2: Mittlere Zeit in Microsekunden zur Schlüsselselektion und Speicherverbrauch

7.2 Speicherverbrauch

Zur Bestimmung der Mindestanforderungen an Geräte auf denen TESLA inklusive einer Anwendung lauffähig ist muss analysiert werden wieviel Speicher die aktuelle Implementierung benötigt. Zur statischen Analyse des Komplilats stellt die *GNU Arm Embedded Toolchain* [1] unter anderem das Programm *arm-none-eabi-size* bereit welches Information über die einzelen Sektion ausgibt. Mit Hilfe eines Shell-Skripts können die relevanten Information extrahiert und formatiert zusammen gefasst werden. Ruft man das Skript auf und übergibt diesem ein aktuelles Komplilat erhält man die im Auszug 7.1 dargestellten Ergebnisse. Bei näherer Betrachtung der Ausgabe fällt auf, dass *.data* in Program und Data einfließt was sich damit begründen lässt, dass dieser Sektor initialisierte Variable beinhaltet welche zur Laufzeit aus dem ROM in den RAM kopiert werden. Die Sektoren *.text* und *.bss* beinhalten hauptsächlich Quellcode, Konstanten sowie nicht initialisierte Variablen die zur Laufzeit mit $'0'$ initialisiert werden.

```
1 ./printmem.sh tesla-riot.elf
2 STM32 Memory Usage:
3 ----------------------
4 Program:    71000 bytes (54.2% Full) (.text + .data + .bootloader)
5 Data:       21724 bytes (265.2% Full) (.data + .bss + .noinit)
```

Listing 7.1: TESLA Kompilat Speicherbedarf

Addiert man beide Werte erhält man ca 93 KB, die die im Kapitel 4 beschriebene Im-

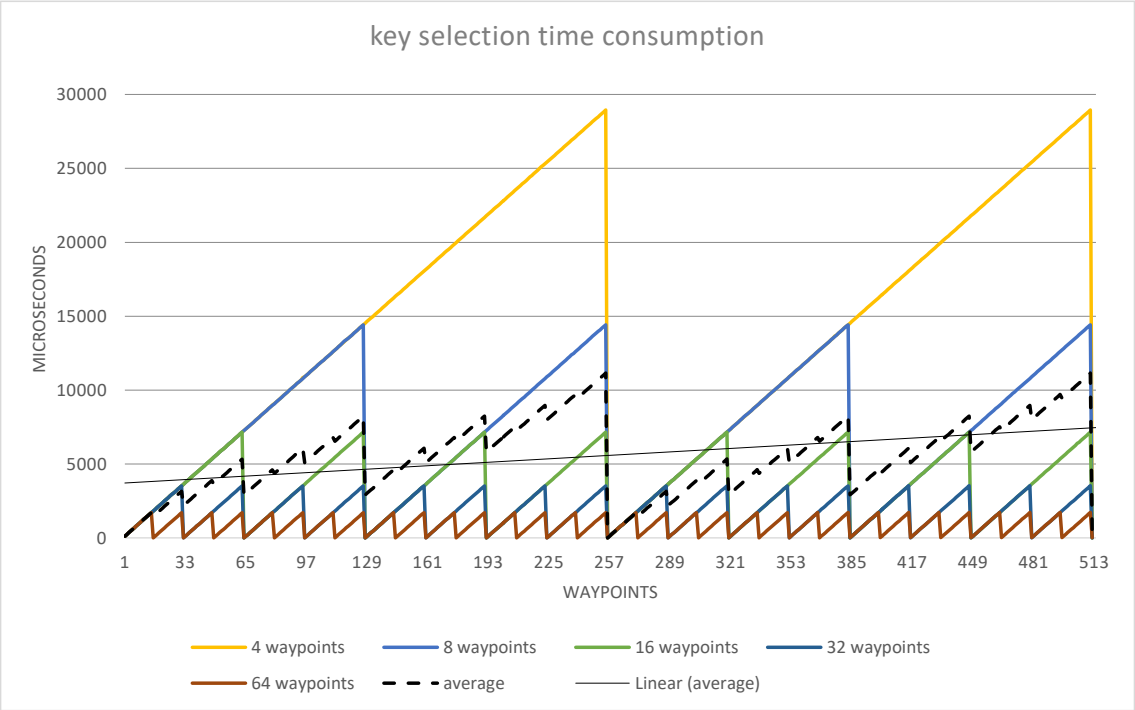

Abbildung 7.3: Zeit in Microsekunden zur Schlüsselselektion in Abhänigkeit der Anzahl von Wegpunkten

plementierung an ROM benötigt. Somit lassen sich Geräte der Klassen 0 für Verwendung ausschließen und die Anforderung stellen, dass mindestens Geräte der Klasse 1 oder besser verwendet werden müssen damit TESLA mit ESP in einer Applikation verwendet werden kann, sofern die maximale Gesamtgröße von 100 KB nicht eingehalten werden kann.
Im Hinblick auf den RAM lassen sich keine genauen Angaben machen, da der benötigte RAM bedingt durch die Konfiguration von TESLA sowie der Größe der Payload abhängt. Es lässt sich jedoch ein Minimalbedarf bestimmen, sofern man die Größe der Payload exemplarisch fixiert. Unter der Annahme das die Intervallverzögerung zur Veröffentlichung eines Schlüssels $d = 2$ und die Größe der Payload 10 KB beträgt sowie auf die Verwendung von Wegpunkte bei der Erzeugung der Key-Chain verzichtet wird, kann ein garantierter RAM Speicherbedarf für ein Paket mit 143 Byte beziffert werden (siehe Tabelle 7.1). Ist davon auszugehen,

	IPv6 Header	40 Byte
+	ESP Header	18 Byte
+	ICV	34 Byte
+	Payload	10 byte
+	TESLA Trailer	41 Byte
=		143 Byte

Tabelle 7.1: Gesamtgröße eines ESP Datagrams

dass mindestens ein Paket pro Zeitinterval verschickt wird, müssen insgesamt drei Pakete im Speicher gehalten werden, wodurch sich ein garantierter Speicherbedarf von $3 * 143 = 429$

Byte ergibt. Nach der Tabelle 2.1 verfügen Geräte der Klasse 1 ca 10 KB RAM und bieten somit ausreichend Kapazitäten um selbst 10 Pakete pro Zeitinterval zwischenspeichern zu können. Damit lässt sich die zuvor gestellte Mindestanforderung zur Verwendung von Geräte der Klasse 1 festigen.

7.3 Testergebnisse

Die Resultate der im Kapitel 5 beschriebenen Testszenarien sollen im Folgenden betrachtet werden um die Eingangs gestellte Frage ob TESLA in einem Smart Home Szenario verwendet werden kann, zu beantworten.

7.3.1 Ein Sender ↔ Ein Empfänger

Dieses Szenario mit nur zwei Kommunikationsendpunkte ist eine Mindestkonfiguration um die Kommunikation zwischen zwei Teilnehmern zu untersuchen. Hierfür wurden beide Endpunkte vollständig mit TESLA initialisiert und anschließend drei Nachrichten verschickt. Die in der Phase *TESLA_DATA* versandten Pakete gingen vollständig beim Empfänger ein und es konnte erfolreich, entsprechend $d = 3$, mit dem dritten eingehenden Paket das Paket des ersten Intervals authentifiziert werden.
Wodurch bestätigt werden kann, dass TESLA in einem einfachen Szenario funktioniert wobei die verwendeten Knoten nicht direkt nebeneinander positioniert sind.

7.3.2 Ein Sender ↔ fünf Empfänger

Im zweite Szenario ist untersucht worden, ob die Zeitverzögerung T_δ ein Indikator dafür ist, dass die Kommunikation zwischen zwei Endpunkten mit TESLA authentifiziert werden kann. Hierzu wurde in der Phase der Zeitsychronisation, die $T_\delta = 95$ gemessen, welcher in weiteren Testdurchläufen leichte Schwankungen von einer MS aufweiste. Da der Wert von T_δ in die Überprüfung (siehe Formel 3.12) der oberen Grenze des Intervals miteinbezogen wird, hatte dieser keine Auswirkungen auf die Funktionalität von TESLA in diesem Test. Außerdem veränderte sicher der Wert von T_δ nicht signifikant bei größeren Abständen zwischen Sender und Empfänger. Es konnte jedoch festgestellt werden, dass die Kommunikation zwischen zwei Geräten ab einem Abstand von 8-10 Knoten nicht mehr möglich ist, was sich durch nicht empfangene Pakete bemerkbar machte. Abschließend zu diesem Test kann ausgeschlossen werden, dass T_δ ein Indikator dafür ist, ob eine Kommunikation mit TESLA authentifiziert werden kann oder nicht. Zusätzlich kann man zu dem Schluss kommen, dass TESLA mit ESP aufbauend auf dem ersten Test mit nur zwei Endpunkten, ebenfalls mit mehreren Endpunkte umgehend kann.

7.3.3 Ein Sender ↔ viele Empfänger

Ein weiteres Testszenario soll zeigen, wie sich die prototypische Implementierung von TESLA mit mehr als fünf Empfängern verhält. Dabei wurde zunächst versucht alle im Experiment eingeschlossenen M3-Knoten gleichzeitig zu initialisieren, was nicht zu einer Überlastung des Senders, allerdings zu Paketverlust führte. Der Verlust von Paketen äußerte sich dahingehend, dass lediglich vier Empfänger erfolgreich initialisiert werden konnten. Dieses Verhalten war reproduzierbar, da bei weiteren Durchläufen die gleichen Empfänger initialisiert worden

sind. Mit dieser Erkenntnis wurde der Test erneut durchgeführt, mit dem Unterschied dass nicht versucht wurde alle Empfänger gleichzeitig zu initialisieren, sondern jeder Einzeln mit einem Zeitversatz von vier Sekunden eine Anfrage zur Zeitsychronisation stellte. Somit war es möglich, erfolgreich mehr als vier Empfänger zu initialsieren und es konnten insgesamt 18 Empfänger initialisiert werden. Bei dem Versuch mehr als 18 Empfänger zu initialiseren, war es dem Sender nicht mehr möglich weitere Anfragen zu verabeiten, da ihm nicht mehr ausreichend Speicher zur Verfügung stand. In einem weiteren Durchlauf konnten alle initialisierten Empfänger Pakete vom Sender empfangen, zwischenspeichern und abschließend erfolgreich authentifizieren.

7.3.4 Maximaler Abstand zwischen Sender und Empfänger

Zur Überprüfung wie groß die Verzögerungen T_δ sein kann, ohne die Kommunikation mit TESLA authentifizierten Paketen zu beeinflussen, wurde ein Test mit einem Sender und einem Empfänger aufgebaute und entsprechend positioniert um auserhalb des jeweiligen Senderadius zu sein. Die Lücke zwischen beiden Senderadien sollte mit Hilfe von als Router aggierenden M3-Knoten gefüllt werden. Leider war es nicht möglich eine Verbindung zwischen Empfänger und Sender aufzubauen, wodurch dieser Test keine weiteren Erkenntnisse liefern kann.

7.3.5 Implementierungskonformität

Zusätzlich zu den im Kapitel 5 beschriebenen Experimenten soll ein weiterer Test die Implementierungskonformität, der im Rahmer dieser Arbeit entstanden TESLA Implementierung, untersuchen. Dafür wurde die Implementierung von Jonas Dellinger, der im Rahmen seiner Arbeiten (siehe Abschnitt 6.6) ebenfalls TESLA prototypisch implementiert hat, herangezogen. Da beide Prototypen unabhängig voneinander und ohne weiterer Absprache entwickelt worden sind, lässt dieser Test eine Aussage darüber zu, ob beide Implementierung konform sind.
Für den Test übernahmen der Prototyp von Jonas Dellinger, der offensichtlich für leistungsfähigere Geräte konzipiert ist, die Rolle des Senders und im Umkehrschluss der Prototyp dieser Arbeiter die Rolle des Empfängers. Bevor untersucht werden konnte, ob beide Implementierungen Daten austauschen können, mussten die Datentypen der TESLA spezifischen C-Strukturen verglichen und angepasst werden, um potenziell Fehlinterpretationen der Daten zu vermeiden. Weiter wurde im Rahmen dieses Abgleichs festgestellt, dass Jonas Dellinger die Antwort zur Zeitsychronisation mit einem Paket abhandelt. Das erforderte eine weitere Anpassung des Prototypen dieser Arbeit um die Senderzeit T_s sowie die Konfigurationsparameter von TESLA in einem Paket verarbeiten zu können. Während des Test sind weitere Unterschiede zwischen den beiden Implementierungen festgestellt worden. Zunächst wurden die ICVs mit verschiedenen Algorithmen berechnet, was sich allerdings durch die Umstellung von SHA-1 auf SHA-256 des TESLA-Linux Prototypen von Jonas Dellingen schnell bereinigen ließ. Desweiteren wurde die Payload in der Initialisierungsphase Verschlüsselt, wodurch der TESLA-Riot-OS Prototyp die Antwort des Senders nicht verarbeiten konnte. Auch dieses Problem hat sich einfach beheben lassen, in dem die Verschlüsselung auf Senderseite deaktiviert wurde. Nachdem in beiden Implementierungen der gleiche PSK vorlag, war es dem Sender und Empfänger möglich miteinander zu kommunizieren. Dank dem erfolgreichen Austausch von Daten, konnte weitere Fehler in der Implementierungen identifiziert

und behoben werden, wodurch es schlussendlich möglich war den Empfänger vollständig zu initialisieren.
Der Versuch Pakete auf Empfängerseite zu authentifizieren war leider ohne Erfolg, da sich die TESLA spezifischen Datenpakete beider Prototypen unterscheiden und somit nicht mit der Verarbeitungslogik des Empfängers übereinstimmt. Es kann jedoch davon ausgegangen werden, dass mit weiteren Anpassungen der Prototypen eine erfolgreiche Authentifizierung der Daten möglich ist. Auf Grund einzuhaltener Abgabefristen, können die Erkenntnisse aus weiteren Tests mit beiden Prototypen nicht mehr in den Ergebnissen dieser Arbeiten dokumentiert werden. Es lässt sich jedoch zusammenfassend ein Teilerfolg zum Test der Implementierungskonformität verzeichnen.

8 Zusammenfassung und Ausblick

Das Ziel dieser Arbeit wurde erreicht und es lässt sich mit den Ergebnissen der Evaluation im Abschnitt 7 bestätigen, dass das Protokoll TESLA zusammen mit ESP in einem Smart Home Szenario funktionieren kann. TESLA konnte erfolgreich in einem Prototypen umgesetzt und auf Geräten der Klasse 1 als Sender sowie als Empfänger verwendeten werden. Desweiteren wurde ebenfalls erfolgreich das Protokoll ESP aus der IPsec Protokoll Familie adaptiert um einen sicheren Kanal zur Initialisierung der Empfänger zu gewährleisten, sowie einen Rahmen zur Übertragung von mit TESLA authentifizierten Daten bereitzustellen. Da zum Zeitpunkt des Erstellens dieser Arbeit keine Implementierung von ESP im GNRC von Riot-OS zur Verfügung stand, stellt die Umsetzung in ESP zunächst ein Problem dar. Dies wurde jedoch im Verlauf der Entwicklung des Prototypen gelöst, in dem ESP zusammen mit TESLA implementiert wurde. Auf Grund der mangelhaften Unterstüzung von IPsec im GNRC, lässt sich jedoch keine Aussage darüber treffen ob die ESP Pakete von allen vollumfänglichen protokollkonformen ESP Implementierungen korrekt interpretiert und verarbeitet werden können, da es nicht möglich war das Feld *Next-Header* im IPv6 Header auf den korrekten Wert des *IPv6 Extention Header* zu setzen. Ein weiteres Problem stellt die Zeitsychronisation dar, da die Spezifikation von TESLA lediglich die Verwendung von idententische Zeiteinheiten vorschreibt. Die verwendeten M3-Knoten messen jedoch die Zeit seit Systemstart wodurch sie keine Unix-Zeitstempel kompatible Zeit zurückgeben können. Dies hat zwar prinzipiell keinen Einfluss auf die Funktionsweise von TESLA, kann jedoch zu Fehlverhalten führen, falls die zu verarbeitenden Zeitstempel zu große Zahlen beinhalten und sich unter Umständen in inkorrekten Werten äußern können.

Obwohl das Protokoll TESLA das fortwährende Problem der Authentifizierung von Multicast-Kommunikation zu lösen scheint, hat das Protokoll Optimierungsbedarf, wie die verschiedenen TESLA Varianten im Kapitel 6 zeigen. Jede dieser TESLA Abwandlung versucht fokusiert eine Problemstellung im Zusammenhang mit der Verwendung von TESLA zu lösen. Ein offentlichtliches Problem von TESLA besteht in der Notwendigkeit, dass alle Empfänger neu initialisiert werden müssen, sobald der letzte Schlüssel einer Key-Chain verwendet wurde. Dadadurch wird eine kontinuierliche und unterbrechnungsfreie Datenübertragung verhindert. In dem Fall, dass sich viele Empfänger zur selben Zeit neu initialisieren müssen, kann das zu einer Überlastung des Senders führen, was einer DoS Attacke ähnelt. Des weiteren ist TESLA anfällig gegenüber speicherorientierten DoS Angriffen, bei denen versucht wird den Speicher durch Zwischenspeicherung von Pakete voll auszulasten um die Verarbeitung weiterer eingehender Pakete zu verhindern. Um diese Probleme des TESLA Protokolls zu beheben, könnte mit einer Kombination der Lösungen, die beiden Varianten infTESLA und TESLA++ bereitstellen, TESLA weiterentwickelt werden. Die Verbesserungen würden die Resistenz gegenüber speicherorientierten DoS Angriffen erhöhen, ein kontinuierliches und unterbrechnungsfreies senden von Daten ermöglichen, sowie die den Speicherbedarf auf Sender und Empfängerseite verringern. Die Weiterentwicklung zur Verbesserung von TESLA könnte man in einer zukünftigen Arbeiten untersuchen sowie in einem RFC protokollieren.

Desweiteren wird in Rahmen dieser Arbeit nicht betrachtet wie das Schlüsselmaterial zur Clientinitialisierung verteilt wird und zu Grunde gelegt, dass das vorhandene Material bereits über einen sicheren Kanal ausgetaucht worden ist. Daraus ergibt sich, dass in einer zukünftigen Arbeit die Verwendung von IKE im Zusammenhang von TESLA evaluiert werden könnte. Eine Spezialisierung von IKE beschreibt *Tobias Heider* in seiner Arbeit *Minimal G-IKEv2 implementation for RIOT OS* [13]. In dieser untersucht Tobias Heider das Gruppenschlüssel-Management Protokoll Group IKEv2 (G-IKEv2) [34], welches auf IKE der Version 2 basiert und die Schlüsselverteilung sowie die Schlüsselverwaltung für Gruppenkommunikation regelt.

Abbildungsverzeichnis

Literaturverzeichnis

[1] Gnu arm embedded toolchain.

[2] E. Baccelli, O. Hahm, M. Gunes, M. Wahlisch, and T. C. Schmidt. Riot os: Towards an os for the internet of things. In *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*, pages 79–80. IEEE, 2013.

[3] C. Bormann, M. Ersue, and A. Keranen. Terminology for constrained-node networks. RFC 7228, RFC Editor, May 2014. `http://www.rfc-editor.org/rfc/rfc7228.txt`.

[4] S. Câmara, D. Anand, V. Pillitteri, and L. Carmo. Multicast delayed authentication for streaming synchrophasor data in the smart grid. In *IFIP International Information Security and Privacy Conference*, pages 32–46. Springer, 2016.

[5] A. Conta, S. Deering, and M. Gupta. Internet control message protocol (icmpv6) for the internet protocol version 6 (ipv6) specification. RFC 4443, RFC Editor, March 2006. `http://www.rfc-editor.org/rfc/rfc4443.txt`.

[6] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile. RFC 5280, RFC Editor, May 2008. `http://www.rfc-editor.org/rfc/rfc5280.txt`.

[7] J. Dellinger. Streaming multicast authentication with tesla and esp on linux.

[8] A. Dunkels, N. Eriksson, F. Österlind, and N. Tsiftes. The contiki operating system. *Web page. Visited Oct*, 24, 2006.

[9] C. Eckert. It-sicherheit, 4. *Aufl., München, Wien*, 2006.

[10] F. I. T. Facility. F.i.t. iot-lab.

[11] F. Forsby, M. Furuhed, P. Papadimitratos, and S. Raza. Lightweight x. 509 digital certificates for the internet of things. In *Interoperability, Safety and Security in IoT*, pages 123–133. Springer, 2017.

[12] D. Gay, P. Levis, D. Culler, and E. Brewer. nesc 1.2 language reference manual, 2005.

[13] T. Heider. Minimal g-ikev2 implementation for riot os, 2017.

[14] J. Ivkovic and J. Ivkovic. Analysis of the performance of the new generation of 32-bit microcontrollers for iot and big data application. In *Proceedings of the International Conference on Information Society and Technology (ICIST), Kopaonik, Serbia*, pages 12–15, 2017.

[15] C. Kaufman. Internet key exchange (ikev2) protocol. RFC 4306, RFC Editor, December 2005. `http://www.rfc-editor.org/rfc/rfc4306.txt`.

[16] S. Kent. Ip authentication header. RFC 4302, RFC Editor, December 2005. http://www.rfc-editor.org/rfc/rfc4302.txt.

[17] S. Kent. Ip encapsulating security payload (esp). RFC 4303, RFC Editor, December 2005. http://www.rfc-editor.org/rfc/rfc4303.txt.

[18] S. Kent and K. Seo. Security architecture for the internet protocol. RFC 4301, RFC Editor, December 2005. http://www.rfc-editor.org/rfc/rfc4301.txt.

[19] H. Krawczyk, M. Bellare, and R. Canetti. Hmac: Keyed-hashing for message authentication. RFC 2104, RFC Editor, February 1997. http://www.rfc-editor.org/rfc/rfc2104.txt.

[20] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[21] M. Malkus. Diet-esp for riot os.

[22] D. McGrew, K. Igoe, and M. Salter. Fundamental elliptic curve cryptography algorithms. RFC 6090, RFC Editor, February 2011. http://www.rfc-editor.org/rfc/rfc6090.txt.

[23] D. Mills, J. Martin, J. Burbank, and W. Kasch. Network time protocol version 4: Protocol and algorithms specification. RFC 5905, RFC Editor, June 2010. http://www.rfc-editor.org/rfc/rfc5905.txt.

[24] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. Pkcs #1: Rsa cryptography specifications version 2.2. RFC 8017, RFC Editor, November 2016.

[25] T. Narten, E. Nordmark, W. Simpson, and H. Soliman. Neighbor discovery for ip version 6 (ipv6). RFC 4861, RFC Editor, September 2007. http://www.rfc-editor.org/rfc/rfc4861.txt.

[26] J. M. Park, E. K. Chong, and H. J. Siegel. Efficient multicast packet authentication using signature amortization. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 227–240. IEEE, 2002.

[27] A. Perrig, R. Canetti, J. D. Tygar, and D. Song. Efficient authentication and signing of multicast streams over lossy channels. In *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*, pages 56–73. IEEE, 2000.

[28] A. Perrig, R. Canetti, J. D. Tygar, and D. Song. The tesla broadcast authentication protocol. *Rsa Cryptobytes*, 5, 2005.

[29] A. Perrig, D. Song, R. Canetti, J. D. Tygar, and B. Briscoe. Timed efficient stream loss-tolerant authentication (tesla): Multicast source authentication transform introduction. RFC 4082, RFC Editor, June 2005.

[30] A. Perrig, R. Szewczyk, J. D. Tygar, V. Wen, and D. E. Culler. Spins: Security protocols for sensor networks. *Wireless networks*, 8(5):521–534, 2002.

[31] Z. Shelby, K. Hartke, and C. Bormann. The constrained application protocol (coap). RFC 7252, RFC Editor, June 2014. `http://www.rfc-editor.org/rfc/rfc7252.txt`.

[32] A. Studer, F. Bai, B. Bellur, and A. Perrig. Flexible, extensible, and efficient vanet authentication. *Journal of Communications and Networks*, 11(6):574–588, 2009.

[33] S. A. Vanstone. Next generation security for wireless: elliptic curve cryptography. *Computers & Security*, 22(5):412–415, 2003.

[34] B. Weis and V. Smyslov. Group key management using ikev2. Internet-Draft draft-yeung-g-ikev2-14, IETF Secretariat, July 2018. `http://www.ietf.org/internet-drafts/draft-yeung-g-ikev2-14.txt`.