FAKULTÄT FÜR
INFORMATIK

Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik

Advanced Multimedia And Security

Verdeckte Kommunikation via Modbus TCP/IP

Abschlussbericht

Autoren:

Sophie Herbrechtsmeyer
Agostino Moosdorf
Jens-Uwe Sinn
Simon Wenk

Betreuer:

Prof. Dr.-Ing. Jana Dittmann
Dipl.-Inf. Robert Altschaffel

Magdeburg, 03.Juli 2019

Inhaltsverzeichnis

Abstract

(Simon)

Diese Arbeit untersucht, wie man Informationen im industriellen Umfeld durch die Kommunikation über das Modbus TCP/IP Protokoll versteckt übertragen kann. Zunächst werden Methodiken, die man dafür anwenden kann, theoretisch beleuchtet. Anschließend werden zwei Methoden, und zwar Artificial Retransmission und Interpacket Times, anhand eines Versuchsaufbaus mit OpenPLC und ScadaBR implementiert und die Praktikabilität und Umsetzungsschwierigkeit evaluiert. Zum Schluss werden Möglichkeiten aufgezeigt, wie man solche steganografischen Angriffe detektieren und gegebenenfalls verhindern kann.

1 Motivation

1.1 Einführung

(Agostino)

Steganografie ist die Wissenschaft, Informationen innerhalb anderer Informationen zu verstecken [CCCK10]. Eine Variante davon beschäftigt sich mit dem verdeckten Mitsenden von Nachrichten in Netzwerkprotokollen.
Die Einbettung versteckter Nachrichten in Protokolle wie TCP ist bereits gut untersucht [MWZ+16].
In dieser Arbeit wird insbesondere auf die Möglichkeiten von Steganografie in Modbus TCP/IP eingegangen. Modbus ist ein serielles Netzwerkprotokoll, welches 1979 von Modicon entwickelt wurde. Es wird im Bereich der Kommunikation von industriellen Geräten wie etwa Speicherprogrammierbaren Steuerungen verwendet. Bei Modbus TCP/IP sind die Daten in ein TCP/IP-Paket eingebettet und die Addressierung erfolgt über IP-Adressen [AMP].

1.2 Aufbau der Arbeit

(Agostino)

Die Projektarbeit untersucht Möglichkeiten, die Implementierung des Modbus TCP/IP Protokolls in OpenPLC zu verändern, um verdeckte Nachrichten zu verschicken. Zunächst werden in Kapitel 2 bereits existierende Ansätze von Steganografie in Netzwerkprotokollen beschrieben und auf die Eigenschaften von Modbus TCP/IP eingegangen. In Kapitel 3.1 werden dann steganografische Ansätze vorgestellt, die auf Modbus TCP/IP zugeschnitten sind und im Rahmen dieser Projektarbeit untersucht und teilweise implementiert werden. Ein von uns angefertigter Demonstrator zeigt zwei Möglichkeiten für steganografische Kommunikation, namentlich ein künstliches Nochmalsenden von Paketen (Artificial Retransmission) und die Manipulation der Absendezeit mit einhergehender Auswirkung auf die Interpacket Arrival Time. Der Demonstrator läuft auf einer Modbus TCP/IP Verbindung in einem Versuchsaufbau, der die Software OpenPLC und ScadaBR benutzt. Eine nähere Beschreibung dieses Versuchsaufbaus sowie der Implementierung des Demonstrators wird in Kapitel 3.2 ausgeführt. Kapitel 4 stellt die Ergebnisse der untersuchten steganografischen Techniken vor. Auf mögliche Detektionsansätze zur Erkennung steganografischer Methoden in Modbus TCP/IP wird in Kapitel 5 eingegangen. In Kapitel 6 wird schließlich das Fazit gezogen und einige offene Fragestellungen werden angeführt. Der im Rahmen dieser Arbeit angefertigte Code liegt im Anhang und wird an den entsprechenden Stellen referenziert.

2 State Of The Art

2.1 Netzwerk-Steganografie

(Uwe)

Unter Steganografie versteht man im Allgemeinen das Verstecken von geheimen Informationen in einer Datei. Bei der Netzwerk-Steganografie geht es darum, diese geheimen Informationen über einen unverdächtigen Kommunikationskanal unbemerkt zu versenden. So können zum Beispiel vertrauliche Daten aus einem geschützten Bereich geschleust werden, ohne dass dies dem zuständigen Sicherheitspersonal auffällt. Hieran sieht man, dass Netzwerk-Steganografie in erster Linie ein Angriff auf die Vertraulichkeit eines IT-Systems darstellt.

2.1.1 Ist-Zustand der Forschung

(Uwe)

In der vorliegenden Arbeit soll die Möglichkeit untersucht werden, verdeckt Informationen in einen Modbus TCP/IP Kanal einzubetten. Typischerweise werden die Verbindungen der ersten und zweiten Ebene des Netzwerk-Stacks (OSI-Schichtmodell) durch Ethernet oder kabellos durch ein Protokoll der IEEE 802.11-Familie beschrieben. Die weiteren Schichten sind dann eine Version des Internetprotokolls, TCP und in der Anwendungsschicht das Modbus-Protokoll. Steganografie lässt sich in jeder dieser Ebenen umsetzen. Untersuchungen bezüglich steganografischer Methoden für alle erwähnten Protokolle außer Modbus finden sich zahlreich in der Literatur. Deshalb wird hier nur auf eine kleine Auswahl verwiesen. Steganografie in TCP und IP wird von Lewis et al. ausführlich behandelt [ML05]. Für einen genaueren Überblick eignet sich das Buch “Information Hiding in Communication Networks“ von Mazurczijk et al. [MWZ+16]. Soweit den Autoren bekannt ist, sind derartige Forschungen für das Modbus-Protokoll nicht publiziert und deshalb konzentriert sich diese Arbeit auf dieses, ohne weitere Protokolle oder das Zusammenwirken mit weiteren Protokollen zu betrachten.

2.1.2 Network Information Hiding

(Uwe)

Netzwerkprotokolle bieten eine Vielzahl von Möglichkeiten, um Informationen verdeckt zu übertragen. Man kann hierbei zwei Kategorien unterscheiden: die Einbettung der geheimen Botschaft in Speicherelemente des Protokolls, oder ihre Einbettung in ein zeitlich angepasstes Sendeverhalten (vgl. Abbildung 2.1). Für Angriffe der ersten Kategorie werden im Folgenden die Möglichkeiten betrachtet, Informationen im Payload und in ungenutzten Header-Feldern zu verstecken. Timing-Channel-Angriffe kann man weiterhin in protocol-aware und protocol-agnostic unterteilen [MWZ+16]. Protocol-aware heißt, dass spezifische Timing-Mechanismen des Protokolls verwendet werden. In dieser Arbeit werden zwei Timing-Channel-Angriffe untersucht und praktisch umgesetzt.

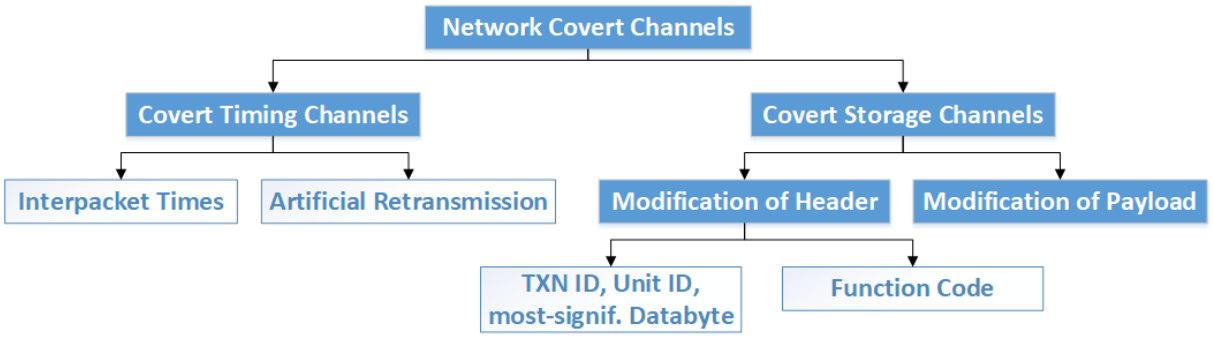

Abbildung 2.1: Steganografische Kommunikationswege in Netzwerken. Abbildung in Anlehnung an: [WZFH15], Seite 13, Fig.2

2.2 Modbus

(Sophie)

Das Modbus-Protokoll ist ein industrieller Standard, der beschreibt, wie diskrete oder analoge I/O Register oder Registerdaten bei der Kommunikation von Industrieanlagen auf Anwendungsebene (Application Layer) organisiert und interpretiert werden [ACR05]. Es gibt neben dem Modbus-Protokoll noch weitere Feldbussysteme, die als Kommunikationsstandards für Industrieanlagen dienen können, auf die in diesem Projekt nicht weiter eingegangen wird.

Der Kommunikation liegt das Master-Slave Modell zugrunde, im Kontext von TCP/IP wird es auch Client-Server Modell genannt. Der Master/Client ist eine Kontrolleinheit, die mit einem oder mehreren Überwachungsgeräten (Slave/Server) verbunden ist. Dabei kann eine Anfrage (Query) ausschließlich vom Master initiiert werden. Das adressierte Slave-Gerät bearbeitet die Anfrage und sendet eine entsprechende Antwort (Response) an den Client zurück. Wie die Daten in den Endgeräten verarbeitet werden, wird nicht durch Modbus, sondern durch die konkrete Implementierung der programmierbaren Steuereinheit (Programmable Logical Controller) bestimmt.

Das Format der Modbus-Anfrage und -Antwort besteht aus vier Feldern: Geräteadresse, Function Code, Datenfeld und Error Checksum (vgl. Abbildung 2.2, obere Zeile “Traditionelles Modbus Paket“). Die Geräteadresse enthält bei der Anfrage die Slave-Geräteadresse, bei einer Antwort die Adresse des Masters. Der Function Code ist eine acht Bit große Zahl

zwischen 0 und 255 und informiert das Slave-Gerät darüber, welche Aktion (ggf. mithilfe der Daten aus dem Datenfeld) ausgeführt werden soll. Der Standard sieht vor, dass beispielsweise eine 0 nur vom Master für Broadcast-Nachrichten verwendet werden darf und bestimmte Function-Code Bereiche bereits definiert sind, andere wiederum individuell vom Nutzer belegt werden können. Geräte können nur die Function-Codes verarbeiten, deren Funktionalität/Verarbeitung in ihrer Programmierung vorgesehen ist. Kann die Anfrage erfolgreich bearbeitet werden, wird der gleiche Function-Code zurückgesendet. Tritt ein Fehler auf, wird der most-significant Bit des Function Codes auf 1 gesetzt und ggf. Informationen über den aufgetretenen Fehler im Datenfeld der Antwort notiert und an den Master zurückgeleitet. Zuletzt dient die Error Checksum sowohl beim Master als auch beim Slave dazu überprüfen zu können, ob die gesendeten Daten fehlerfrei (ohne Verlust) übertragen wurden, sodass bei einem Fehler ggf. vom Master eine neue Query initiiert wird. Der Master schickt niemals zwei Anfragen nacheinander an das gleiche Slave-Gerät, ohne dessen Antwort abzuwarten oder bis ein Timeout überschritten wurde.

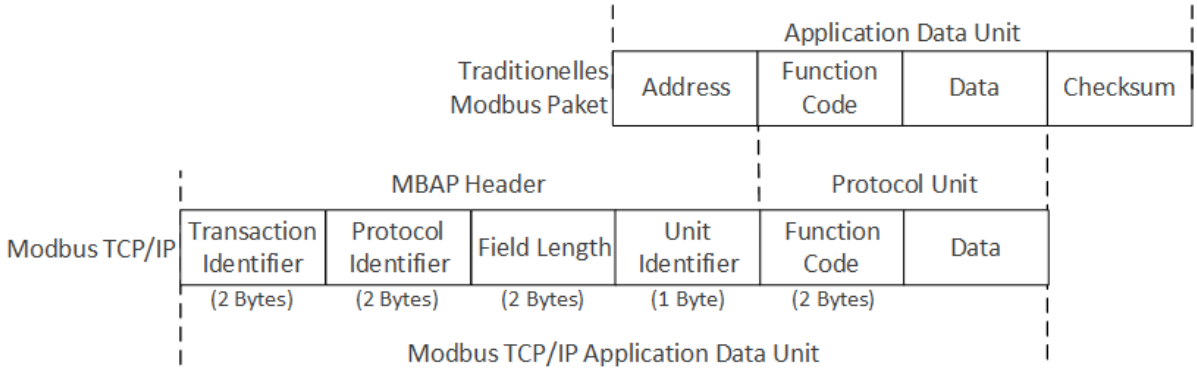

Abbildung 2.2: Konstruktion eines Modbus TCP/IP Pakets, in Anlehnung an [ACR05]

2.2.1 Modbus via TCP/IP

(Sophie)

Für das traditionelle Modbus werden serielle Schnittstellen wie EIA-232 und EIA-485 genutzt. Dagegen wird im Anwendungsfall von Modbus TCP/IP auf der Transportebene (Transport Layer) das weit verbreitete Netzwerkprotokoll TCP/IP genutzt, um den Datenaustausch über Infrastrukturen zu nutzen, die für Ethernet oder WLAN vorgesehen sind.

Für die Übertragung wird der traditionelle Aufbau des Modbus-Pakets aufgebrochen und in das Modbus Application Protocol (MBAP) für TCP/IP eingebettet ([ACR05], vgl. Abbildung 2.2). Adressfeld und Error Checksum entfallen, dafür besteht das Paket nun aus dem MBAP-Header, dem Function Code und dem Datenfeld. Abbildung 2.2 zeigt den Aufbau des Modbus-Pakets und die Byte-Größe der Felder. Das Feld Transaction Identifier ermöglicht die eindeutige Zuordnung des Pakets zu einer Transaktion, wenn der Client inhaltlich verschiedene Anfragen an einen oder mehrere Server sendet. Es gibt keine zusätzlichen Bedingungen. Die Protocol ID ist für Modbus immer 0, andere Zahlen sind für mögliche Erweiterungen reserviert. Das Modbus-Paket wird nicht als Modbus-Service erkannt, wenn dieses Feld eine andere Zahl außer 0 enthält und somit nicht bis

zum Application Layer weitergeleitet. Es werden zwei Byte für die Angabe der Datenlänge des Modbus-Pakets reserviert. Das erste Feld repräsentiert die most-significant Bytes zur Angabe der Datenlänge. Da die Konvention eingeführt wurde, dass ein Modbus-Paket nicht länger als 256 Byte sein darf, wird dieses Feld nicht mehr benötigt. Das zweite Feld zur Angabe der Datenlänge muss korrekt angegeben werden, da sonst das Modbus-Paket nicht erkannt und somit nicht zum Application Layer des Empfängers weitergeleitet wird. Der Unit Identifier wird genutzt, um ein Gerät im Modbus-Netzwerk anzusprechen, das hinter dem Empfänger liegt und, wie beim traditionellen Modbus, nicht über TCP/IP kommuniziert. Ist dies nicht der Fall, repräsentiert das Feld eine 0. Das Feld Function Code gibt an, welche Aktion der Server ausführen soll und kann, je nach Applikation, bis zu 256 Function Codes nutzen, wobei beispielsweise die 0 nur für Broadcasting seitens des Clients genutzt wird und bestimmte Wertebereiche für individuelle Funktionalitäten reserviert sind.

Für die Antwort des Servers wird der Modbus-Header der Anfrage vollständig kopiert und das zweite Feld der Datenlänge entsprechend der zu sendenden Daten angepasst. Der Function Code wird bei erfolgreicher Verarbeitung kopiert, oder beim Auftreten eines Fehlers das most-siginificant Byte auf 1 gesetzt, um auf den Fehler hinzuweisen. Im Dezimalsystem erhöht sich somit der Wert des Function Codes um 128. Zudem wird ggf. der Inhalt des Datenfeldes angepasst, um den Fehler näher zu spezifizieren.

3 Theorie und Praxis

3.1 Theorie

3.1.1 Network Information Hiding: Payload-Modifizierung

(Simon)

Bei der Payload-Modifizierung sind die Angriffsvarianten breitgefächert. Darüber hinaus können viele Daten pro Paket verschickt werden. Die Grenze dafür bildet lediglich das Protokoll, welches die maximale Größe eines Paketes definiert: Das Paket darf 256 Byte nicht überschreiten. Zieht man davon 8 Byte, die für Header und Function Code reserviert sind, ab, bleiben noch 248 Bytes. Versteckt man die Informationen geschickt, ist der Detektionsaufwand sehr hoch.

Wie viel übertragen werden kann und wie man vorgeht, hängt von der Anwendung ab. Das heißt auch, man benötigt Hintergrundinformationen über das System und wie dieses genutzt wird.

Werden beispielsweise Zeitstempel übertragen, die durch eine Genauigkeit im Picosekundenbereich abgebildet werden, könnte man das least-significant Bit für die Informationsübertragung nutzen, indem man es verändert. Trotz der Verfälschung ist für die meisten Applikationen irrelevant, ob ein Ereignis eine Picosekunde früher oder später stattgefunden hat, diese Schwankungen können auch Messfehlern zugeschrieben werden. Daher ist eine Beeinträchtigung des Systems dadurch gering und wenn überhaupt sehr unauffällig.

In einem ähnlichen Szenario werden Sensorwerte ausgelesen und übertragen. Sofern diese hochauflösend genug sind, kann man ähnlich vorgehen, wie beim Zeitstempel: Man manipuliert das least-significant Bit für den Sensorwert. Hier gilt, dass man aufgrund der Ungenauigkeit von Messinstrumenten Informationen verstecken kann.

Es ist auch denkbar, die ersten zwei oder drei least-significant Bits zu nutzen, um die Information zu verstecken. Gleichzeitig wird Redundanz eingebaut, sodass für mehrere Werte je eine Null beziehungsweise eine Eins gilt und (je nach dem, was für Wert übertragen werden soll) zufällig aus der entsprechenden Wertmenge ein Wert ausgewählt wird. Zwar muss sowohl der Sender als auch der Empfänger Kenntnis von dieser Heuristik haben, jedoch wird durch dieses Zufallsprinzip die Nachricht noch besser versteckt. Daher ist die Übertragung hier noch unauffälliger. Allerdings muss man abwägen, ab wann man die Werte so stark verändert, dass es sichtbare Auswirkungen auf das laufende System hat.

In unserem Versuch werden Registerwerte gesetzt beziehungsweise ausgelesen. Eine weitere Methode dazu, Informationen versteckt zu übertragen, wäre ungenutzte Register zu

verwenden.

Alle vorangegangenen Überlegungen gehen mit der Vermutung einher, dass die Payload-Daten strukturiert sind. Bei unstrukturierten Daten könnte das Verstecken der Informationen noch einfacher sein, da hinter dem Aufbau dann keine konkrete Logik ist und daher eine Analyse des Payloads erschwert.

3.1.2 Network Information Hiding: Modifizierung des Modbus-Headers

(Uwe)

Der Header besteht aus 7 Bytes und einem weiteren Byte für die Funktionscodes. Diese sieben Bytes werden bei der Antwort des Servers kopiert und sind bei korrekter Umsetzung somit nur zur Übertragung versteckter Daten vom Client zum Server und nicht in die andere Richtung geeignet. Bei nicht korrekter Umsetzung ergeben sich für die Serverantwort die gleichen Möglichkeiten wie für die Client-Anfrage. Die ersten beiden Bytes sind der Transaction Identifier. Dieser Wert dient der Zuordnung des Protokolls zu einer bestimmten Anfrage und kann vom Client freigewählt werden. Die Bytes drei und vier sind der Protokoll Identifier, alle Bits sind auf eins gesetzt. Wireshark erkennt ein Paket nicht als Modbus-Paket, wenn der Protokoll Identifier nicht korrekt ist. Daher ist dieses Feld ungeeignet. Die Payload-Länge wird durch die darauffolgenden Bytes bestimmt. Da der Payload nicht länger als 256 Bytes sein darf, wird nur das Byte 6 verwendet. Das Byte fünf soll auf null gesetzt werden. Da es zur korrekten Funktion von Modbus nicht benötigt wird, lässt es sich für steganografische Zwecke kapern. Falls weitere Modbus-Knoten hinter dem Adressierten sind, die nicht über über Modbus TCP/IP angesprochen werden sondern über einen seriellen Bus etc. wird das siebte Byte zu deren Identifikation verwendet. Wenn alle Kommunikationsteilnehmer nur Modbus TCP/IP sprechen, soll das Byte auf null gesetzt werden. Mit dem Funktionscode wird dem Server mitgeteilt, welche Aktion es ausführen soll: Register lesen, Register schreiben etc. Außerdem wird er bei der Serverantwort kopiert und zurückgesendet, bzw. wenn ein Fehler eingetreten ist, der dazugehörige Fehlercode. Es gibt fest definierte Codes, aber auch die Möglichkeit selbst welche zu bestimmen. So lässt sich auch hiermit Steganografie umsetzen. Abbildung 3.1 zeigt zusammenfassend durch ein Ampelsystem, welche Header-Felder sich zur Steganografie eignen (rot - ungeeignet, gelb - bedingt geeignet, grün - geeignet).

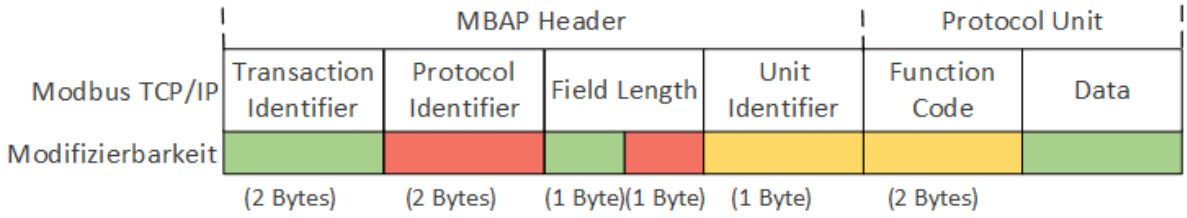

Abbildung 3.1: Steganografie im Modbus-Header, Grafik in Anlehnung an [ACR05].

3.1.3 Network Information Hiding: Verdeckter Zeitkanal

(Sophie)

Steganografie über Modbus TCP/IP wird durch unterliegt verschiedenen Restriktionen, die durch das Client-Server-Modell bedingt werden. Untersucht wird die verdeckte Kommunikation, bei der von einem korrumpierten Server eine versteckte Nachricht an den Client gesendet wird. Da Server nur auf Anfragen (Query) antworten und nicht selbstständig eine Transaktion beginnen dürfen, hängt die zeitliche Komponente vom Client ab. Zunächst ist festgelegt, dass auf eine Query erst eine Response folgen muss, bevor erneut eine Query vom Client initiiert wird. Die Frequenz, mit der solche Modbus-Pakete gesendet werden, ist abhängig von der Pollingrate, welche die Applikation vorgibt und von der Round Trip Time (RTT). Die RTT gibt an, wie viel Zeit benötigt wird, um die Anfrage zu senden, beim Server zu verarbeiten und eine Antwort zu erhalten. Zudem wird ein vom Client vorgegebenes Timeout-Fenster berücksichtigt, sodass beim Überschreiten die Anfrage verworfen und eine neue initiiert wird.

Eine theoretische Überlegung besteht darin, eine versteckte Nachricht mithilfe der Interpacket Times zu übertragen. Die Interpacket Time gibt in diesem Fall an, wie viel Zeit zwischen dem Versenden des Query-Modbus-Pakets und dem Erhalten der zugehöriges Response-Modbus-Pakets vergeht. Es ist möglich, die Antwortzeit des Servers künstlich hinauszuzögern, solang die Antwort innerhalb des vom Client vorgegebenen Timeout-Intervalls folgt (vergleiche Abbildung 3.2). Zum einen wird vorausgesetzt, dass die RTT relativ robust sein muss. Das bedeutet insbesondere, dass die Übertragungsdauer der Pakete in beide Richtungen und die Verarbeitungsdauer der Nachricht beim Server möglichst gleichbleibend sind und nur leichten Schwankungen unterliegen dürfen. Zum anderen muss auf Seiten des korrumpierten Servers das Timeout-Intervall bekannt sein, um Anfragen rechtzeitig zu beantworten. Sonst würden Fehlermeldungen seitens des Clients auftreten, die auf die verdeckte Kommunikation aufmerksam machen. Die Länge des Polling-Intervalls, nach dessen Ablauf eine neue Anfrage vom Client gesendet wird, ist jedoch nicht für die Kodierung relevant, da nur der zeitlichen Abstand zwischen zusammengehörigen Modbus-Paketen gemessen wird. Abhängig von der RTT und der Größe des Timeout-Intervalls kann die Zeitspanne dazwischen in beliebig viele, jedoch mindestens zwei Zeitintervalle unterteilt werden, um mit einer Response mindestens ein Bit der verdeckten Kommunikation zu übertragen. Desto kleiner die Intervalle werden, desto schwieriger wird es, die verdeckte Nachricht bei Schwankungen in der Übertragungsdauer korrekt zu dekodieren.

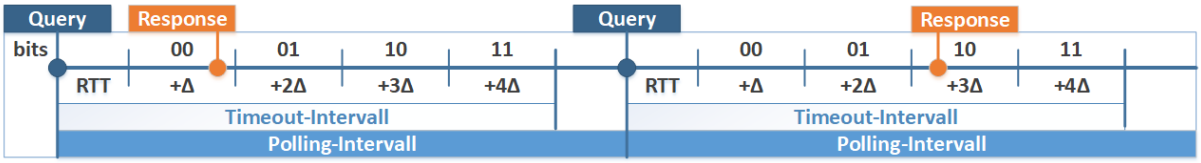

Abbildung 3.2: Steganografie via Interpacket Times, Bit-Kodierung erfolgt über das Hinauszögern der Antwort.

Eine weitere Möglichkeit für Steganografie über den Timing Channel von Modbus TCP/IP sind Artificial Retransmissions [MWZ+16], Seite 69. Dazu sendet der Server mehrfach die gleiche Response auf eine Anfrage des Clients. Die Anzahl der Responses kodiert die Bits der verdeckten Nachricht.

3.1.4 Steganografischer Ablauf

(Agostino)

Abbildung 3.3 zeigt ein mögliches Modell, welches das Verschicken einer Nachricht über einen steganografischen Kanal in folgende logische Schritte unterteilt:
a) Nachricht einlesen: Die zu versendende Nachricht wird eingelesen.
b) Codieren: Die eingelesene Zeichenfolge wird z.B. zu einem Bitstream codiert.
c) Segmentieren: Je nach Art des steganografischen Kanals bzw. dessen Implementierung ist die Anzahl übertragbarer Symbole pro Übertragungsvorgang begrenzt.
Der Bitstream muss daher vor dem Verschicken entsprechend segmentiert werden.
d) Einlesen in den steganografischen Kanal: Eine Modifikation des Codes auf Seite des Senders der zu verschickenden, versteckten Nachricht liest das Segment ein und führt bei Bedarf das für den Kanal charakteristische, geänderte Übertragungsverhalten durch.
e) Auslesen aus dem steganografischen Kanal: Je nach Art des Kanals kann dies bei einem speziellen Empfänger oder auch bei jedem Zwischenhop, der den Netzverkehr mitschneiden kann, passieren.
f) Zusammensetzen: Die einzelnen Segmente werden zu einem Bitstream zusammengefügt.
g) Decodieren: Der Bitstream wird zur ursprünglichen Zeichenfolge decodiert.
h) Nachricht auslesen: Die decodierte Nachricht wird ausgegeben.

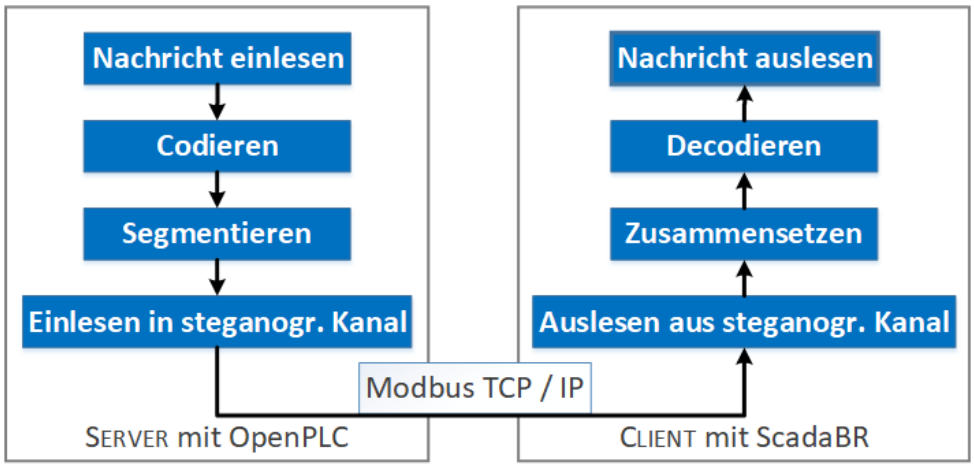

Abbildung 3.3: Steganografischer Ablauf

3.2 Praxis

(Simon)

Um die Steganografie mit dem Modbus TCP/IP-Protokoll umzusetzen, benötigen wir einen programmierbaren logischen Controller (PLC), der über Modbus TCP/IP mit einem anderem Gerät, zum Beispiel einer Mensch-Maschine-Schnittstelle (HMI - Human Machine Interface) kommuniziert. Kompromittiert werden kann dabei sowohl der PLC als auch die HMI, um den steganografischen Angriff durchzuführen; jedoch haben wir uns auf die Modifizierung vom PLC konzentriert.

Es gibt verschiedene PLCs bzw. HMIs. Für das Projekt wurde empfohlen, die quelloffene Software von openplcproject.org zu verwenden. Das Softwarepaket beinhaltet die OpenPLC Runtime, ScadaBR und einen OpenPLC Editor. Die Runtime ist das Herzstück von OpenPLC. Es muss auf dem Gerät installiert sein, auf dem PLC-Programme ausgeführt werden sollen. ScadaBR ist ein HMI Builder. Mit diesem Programm kann man Schnittstellen zwischen Maschine und Mensch bauen und den Zustand der Prozesse visualisieren. Mit Hilfe des OpenPLC Editors kann man PLC-Programme schreiben.

3.2.1 Umsetzung in OpenPLC und ScadaBR

(Simon)

Die OpenPLC Runtime läuft auf einem Raspberry Pi und ScadaBR als virtuelle Maschine (VM) auf einem PC (vgl. Abbildung 3.4). PC und Pi sind physikalisch mit dem Netzwerk verbunden; die VM über eine Netzwerkbrücke. Auf dem PC zeichnet Wireshark den Netzwerkverkehr auf. Alternativ könnte Wireshark auch auf dem Raspberry Pi laufen, da der Netzwerkverkehr auf dem gesamten Kommunikationsweg abgegriffen werden kann; jedoch greifen wir in unserem Szenario die Informationen auf dem PC ab. ScadaBR und Wireshark kommunizieren über das Modbus TCP/IP-Protokoll. Dabei stellt ScadaBR eine Anfrage (um einen Wert auszulesen oder zu setzen) und OpenPLC sendet eine (gegebenenfalls manipulierte) Anfrage zurück. Da TLS in OpenPLC nicht implementiert ist, werden die Daten unverschlüsselt übertragen.

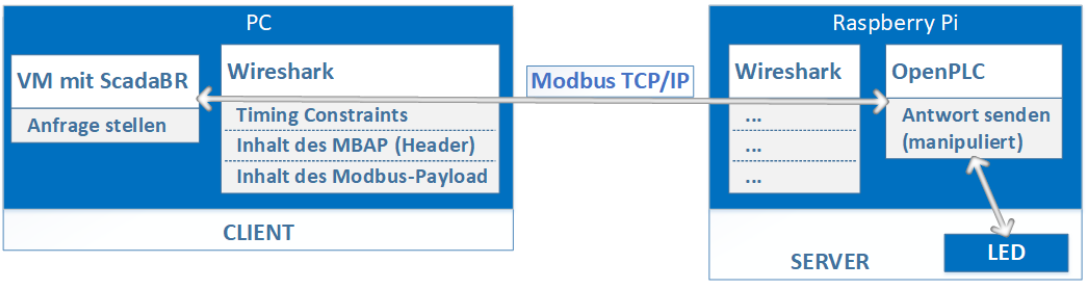

Abbildung 3.4: Versuchsaufbau

Wir haben ein PLC-Programm in Ladder-Logik geschrieben, welches eine LED alle zwei Sekunden ein- und ausschaltet (siehe Anhang). Zur Visualisierung steuert die OpenPLC Runtime eine an einem vom Raspberry Pi bestimmten PIN angeschlossene LED. In unserem Szenario lesen wir die Werte der LED mit ScadaBR aus. Dort haben wir die Datenquelle so eingestellt, dass die Updateperiode (Polling-Intervall) zum Auslesen der Daten bei einer Sekunde und der Timeout bei 150 Millisekunden liegt. Außerdem haben wir den Transporttyp auf “TCP with keep alive“ gestellt. Dies bewirkt, dass die TCP-Verbindung offen bleibt, auch wenn nichts gesendet wird. Dies ist in unserem Fall sinnvoll, da die Updateperiode sehr klein ist und das häufige Öffnen und Schließen einer TCP-Verbindung einen großen Overhead erzeugen würde. Das heißt aber nicht, dass unser Angriff mit “nur“ TCP nicht funktionieren würde. Alle anderen Felder haben wir auf den Standardeinstellungen belassen. (Natürlich muss man auch die korrekte IP zu ScadaBR eintragen.)

Wichtig zu erwähnen ist, dass wir in unserem Versuchsaufbau nur binäre Werte übertragen, also Nullen oder Einsen. Prinzipiell kann man aber auch Werte beliebiger Datentypen austauschen. Dies würde noch mehr Möglichkeiten verschaffen, Informationen in der Payload des Modbus-Protokolls zu verstecken. Da unser Fokus auf der Implementierung der Steganografie über den Timing Channel liegt und die Umsetzung des Projekts einem zeitlichen Rahmen unterliegt, wurde auf die Umsetzung einer Schaltung mit einem komplexeren Datenaustausch verzichtet.

Probleme

(Simon)

Da das Projekt zeitlich begrenzt ist, haben wir uns immer wieder neu ausrichten müssen. Wenn etwas zu lange gedauert hätte, haben wir es verworfen und uns eine vereinfachte Variante überlegt.

Beispielsweise wollten wir zunächst das Projekt komplett auf Softwarebasis umsetzen. Dies war uns allerdings nicht möglich, da es keine passende Software für die Emulierung eines PLC-Controllers gab und ScadaBR sich nicht ohne Weiteres auf einem System installieren ließ. Stattdessen wurde von den Entwicklern eine VM angeboten, die wir dann auch genutzt haben.

Da wir nun zusätzliche Hardware benötigten, haben wir uns entschieden, einen Raspberry Pi als PLC-Controller zu verwenden.

Als nächstes bestand das Problem der Kommunikation. Da im Netz der Universität teilweise virtuellen Maschinen nur IPv6-Adressen zugeordnet wurden; die Software mit IPv6-Adressen jedoch nicht funktionierte, haben wir auf einem eigenständigen Switch als Netzwerkbrücke zurückgegriffen.

Nachdem die Software lief und die Kommunikation sichergestellt war, galt es, das Hello-World-Programm von openplcproject.org zum Laufen zu bringen. Dieses Programm beinhaltet einen Knopf und eine LED. Drückt man den Knopf, schaltet sich die LED ein; lässt man ihn los, dauert es noch zwei Sekunden, bis sich die LED wieder ausschaltet. Da hier mehrere Probleme aufeinandertrafen, haben wir das Programm vereinfacht, indem sich die LED automatisch alle zwei Sekunden ein- und ausschaltet.

Da unser Versuchsaufbau zunächst nicht funktionierte, dachten wir, es läge an der Slave-ID. Da wir keine Informationen hatten, welche Slave-ID dem Raspberry Pi zugeordnet ist, nahmen wir erfolglos einen ESP8266 als Slave-Device in Betrieb.
Es stellte sich heraus, dass dem Raspberry Pi (dem PLC-Controller) die Slave-ID “1“ zugeordnet ist. ScadaBR ist leider keine gut dokumentierte Software; daher mussten wir sie teilweise empirisch erproben.

Zuletzt ergab sich ein Problem mit dem OpenPLC Editor. Wenn man dort ein Programm in Ladder Logic schreibt, kann man dort Variablen unter anderem je eine IEC-Adresse und einen Anfangswert vergeben. Wenn man jedoch beide Felder gleichzeitig ausfüllt, wird das Programm nicht mehr kompiliert. Wir vermuten, dass es sich in dem Editor um einen

Bug handelt. Es scheint, dass es ein Problem mit Raspberry Pi typischen Adressen gibt. Denn bei anderen Beispielprogrammen (mit anderen IEC-Adressen) bestand das Problem nicht. An diesem Punkt muss man wissen, dass man im OpenPLC Editor ein Programm in Ladder Logic schreiben kann, dieses aber vom Editor für die OpenPLC Runtime jedoch in Structured Text übersetzt werden muss, bevor man es dieser übergeben kann. Da das direkt in Structured Text geschriebene Programm von der OpenPLC Runtime ohne Probleme genommen wurde, sind wir zu dem Schluss gekommen, dass der Übersetzungsprozess von Ladder Logic in Structured Text im OpenPLC Editor fehlerhaft zu sein scheint.

3.2.2 Code

(Agostino)

Der von uns implementierte Demonstrator ist modular aufgebaut und orientiert sich an dem in Kapitel 3.1.4 vorgestellten Modell eines steganografischen Ablaufs. Abbildung 3.5 zeigt, welcher Teil des Demonstrators welche Schritte des Modells umsetzt. Die Funktion der einzelnen Programmteile wird im Folgenden beschrieben. Der gesamte Programmcode sowie eine Kurzanleitung zur Benutzung befinden sich im Anhang.

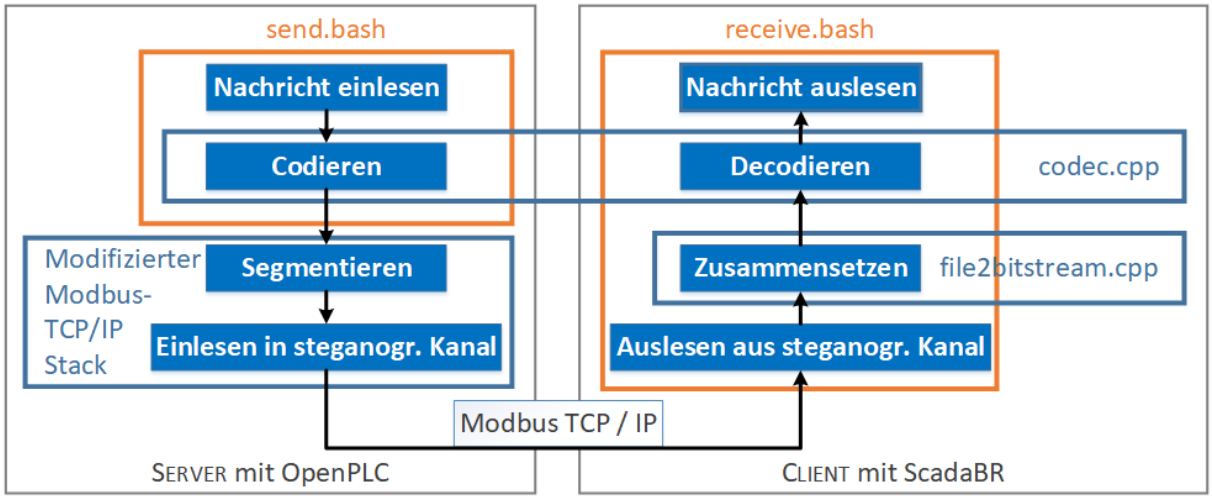

Abbildung 3.5: Steganografischer Ablauf mit zugehörigen Codeteilen.

- *send.bash*: Liegt in unserem Versuchsaufbau auf dem Rechner, auf dem OpenPLC läuft. Liest eine Nachricht aus der Standardeingabe oder aus einer angegebenen Datei und benutzt den Codierer, um diese Nachricht in einen Bitstream umzuwandeln. Dieser Bitstream wird in eine temporäre Datei geschrieben, um dann vom modifizierten Modbus-TCP/IP-Code in der Codebasis von OpenPLC aufgerufen zu werden.
- *codec.cpp*: Bietet Funktionen, um einen String nach einer ausgewählten Zeichencodierung zu einem Bitstream zu codieren. Implementiert ist nur eine Teilmenge des ASCII; ein Hinzufügen anderer Codes ist durch den generischen Aufbau aber

leicht möglich. Außerdem wird beim Vorgang der Codierung eine auswählbare Start-Bitsequenz vorangestellt, sowie die Gesamtgröße des Bitstreams ermittelt und ebenfalls vorangestellt. Diese Informationen geben dem Empfänger die Möglichkeit, Beginn und Ende einer übertragenen Nachricht zu erkennen. Das Überprüfen, ob ein Bitstream eine vollständige Nachricht darstellt sowie umgekehrte Weg der Decodierung eines Bitstreams zu einem String wird durch diesen Teil des Demonstrators auch ermöglicht. Ein Beispiel für die Struktur eines von der ursprünglichen Nachricht konvertierten Bitstreams wird in Abbildung 3.6 angeführt. Hier wurde die Nachricht "text", mit einer spezifizierten Startsequenz "1111", automatisch generierten Informationen über die Nachrichtengröße sowie einer ASCII-codierten Zeichenfolge in eine Bitsequenz zusammengefügt.

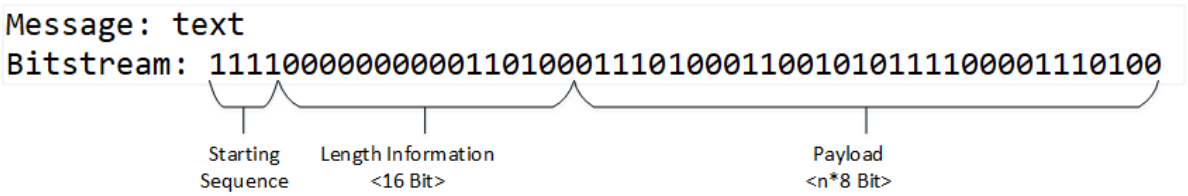

Abbildung 3.6: Beispiel für Bitstream

- Modifizierung des Modbus-TCP/IP-Netzwerkstacks von OpenPLC: Je nach genutzem steganografischen Kanal werden unterschiedliche Teile des Modbus-TCP/IP-Stacks manipuliert. Der modifizierte OpenPLC-Code liest schrittweise Segmente der Bitsequenz aus der temporären Datei aus, in die sie von *send.bash* geschrieben wurde, und in den spezifizierten steganografischen Kanal ein.
- *file2bitstream.cpp*: Aus dem mitgeschnittenen Netzverkehr wird, je nach gewähltem steganografischen Kanal, ein Bitstream extrahiert.
- *receive.bash*: Der Empfänger schneidet den Netzwerkverkehr mit, benutzt die Funktion aus *file2bitstream.cpp*, um einen Bitstream zu extrahieren, und wandelt diesen dann mittels der Funktionen aus *codec.cpp* in die ursprüngliche Zeichenfolge um.

4 Ergebnisse

(Sophie)

In diesem Kapitel werden die Ergebnisse der Projektarbeit vorgestellt. Dazu werden die Bitraten der jeweiligen steganografischen Kommunikationswege aufgezeigt. Die Bitrate gibt an, wie viele Zeichen (und damit Teile der Nachricht) mithilfe des verdeckten Kommunikationskanals innerhalb einer Sekunde übertragen werden können. Die Bitraten werden benötigt, um die implementierten Konzepte miteinander zu vergleichen.Da die Bitraten abhängig vom konkreten Versuchsaufbau sind, werden untere und obere Grenzen der Raten untersucht.
Insbesondere bedingt das Polling-Intervall, d.h. der Zeitabstand, in dem eine neue Query gesendet wird, wie viele Nachrichten in einer Sekunde ausgetauscht werden. Dementsprechend werden maximale Bitraten bei einem minimalen Polling-Intervall erreicht, da der Austausch von Modbus-Paketen häufiger stattfindet. Die Untergrenze des Polling-Intervalls hängt von der Round Trip Time des konkreten Versuchsaufsbaus. Im Versuchsaufbau der Projektarbeit wird bei der RTT eine durchschnittliche Dauer von $0.03s$ beobachtet, mit der im folgenden gerechnet wird, um die maximalen Bitraten zu ermitteln.

4.1 Payload-Modifizierung

Die Payload-Modifizierung wurde in dieser Projektarbeit aus den in Abschnitt **??** genannten Gründen nicht umgesetzt. Die möglichen Bitraten werden dennoch vorgestellt, um sie mit den Ergebnissen der Implementierung vergleichbar zu machen.
Die Bitrate berechnet sich durch die folgende Formel:

$$Bitrate(Payload) = \frac{n\ bit}{Pollingintervall}$$

Sie ist von dem Inhalt und der Größe der Nutzlast abhängig. Im Versuchsaufbau werden nur Modbus-Pakete gesendet, deren Payload eine 0 oder 1 enthält. Somit kann nur ein Bit pro Query-Response-Paar übertragen werden und so die untere Grenze der Payload-Modifizierung bestimmt. Die obere Grenze wird durch die maximale Payload-Größe von 248 Byte bedingt, wobei die verdeckte Nachricht allerdings als Klartext im Payload notiert werden würde.

$$Bitrate(untereGrenze) = \frac{1bit}{0.03s} = 33.3\frac{bit}{s}$$

$$Bitrate(obereGrenze) = \frac{1984bit}{0.03s} = 66133.3\frac{bit}{s}$$

4.2 Header-Modifizierung

Die Header-Modizifierung wurde ebenfalls nur theoretisch vorgenommen. Werden ausschließlich Function-Codes modifiziert, steht im zugehörigen MBAP-Header ein Feld mit der Länge von acht Bit zur Verfügung. Somit ergibt sich eine maximale Bitrate von $266.7bit/s$.

$$Maximale\ Bitrate(Function\ Codes) = \frac{Bits}{Pollingintervall} = \frac{8bit}{0.03s} = 266.7\frac{bit}{s}$$

Betrachtet man alle Header-Felder (ausgenommen der Function Codes), muss berücksichtigt werden, welche Felder nicht verändert werden können, da ansonsten die Zustellung des Modbus-Pakets an den Client nicht mehr möglich ist. Das betrifft die Felder Protocol ID und das zweite Feld zur Angabe der Paketlänge (Databyte). Folgende Felder können modifiziert und damit die entsprechende Feldlänge genutzt werden: Transaction ID (2 Byte), most-significant Databyte (1 Byte), Unit ID (1 Byte). Daraus ergeben sich 4 Byte, also 32 Bit, die zur Übertragung genutzt werden können und somit eine maximale Bitrate von $1066.7bit/s$

$$Maximale\ Bitrate(Header\text{-}Felder) == \frac{Bits}{Pollingintervall} = \frac{32bit}{0.03s} = 1066.7\frac{bit}{s}$$

4.3 Artificial Retransmission

Bei der Artificial Retransmission wird nur ein Bit pro Nachricht kodiert, denn entweder erfolgt nur eine Response (kodiert 0), oder eine doppelte Response (kodiert eine 1). Somit erreicht die Artificial Retransmission eine maximale Bitrate von $33.3bit/s$.

$$Maximale\ Bitrate = \frac{Bits}{Pollingintervall} = \frac{1bit}{0.03s} = 33.3\frac{bit}{s}$$

4.4 Interpacket Times

Bei der Interpacket Time ist zu beachten, dass die Antwortzeit und somit auch die RTT künstlich hinausgezögert wird. Dadurch wird nicht mit dem Polling-Intervall als Untergrenze, sondern mit der durchschnittlichen Dauer einer verzögerten Antwort gerechnet, denn es wird angenommen, dass die Bit-Kodierung uniformverteilt ist. Da das Hinauszögern der Antwort abhängig vom Timeout-Intervall der Applikation ist, wird für die Berechnung ein Timeout von 150ms wie in der Projektarbeit verwendet. Zudem wird das gesamte Intervall nur in zwei Bereiche geteilt, sodass nur 1 Bit pro Nachricht übertragen werden kann.

$$Bitrate = \frac{Bits}{((Timeout - RTT) * 0.5) + RTT}$$

$$Maximale\ Bitrate = \frac{1bit}{((0.15s - 0.03s) * 0.5) + 0.03s} = 11.1\frac{bit}{s}$$

5 Detektion und Vermeidung

(Uwe)

Dieses Kapitel beschäftigt sich mit Gegenmaßnahmen, um die beschriebenen Angriffe zu erkennen und weitestgehend unschädlich zu machen. Verdeckte Kommunikationskanäle können oft nicht komplett geschlossen werden [MWZ+16]. Daher ist eine sinnvolle Strategie ihre Kapazität so weit wie möglich zu verringern.

Header Information Hiding

Die Header-Bytes 3, 5 und 7 haben Sollwerte, die durch eine Stateless-Firewall validiert werden können. Eine Stateful-Firewall kann außerdem prüfen, ob eine Server-Antwort auf zu einer Client-Anfrage passt, d.h. ob alle Felder des Antwortheaders denen des Anfrage-Headers entsprechen. Der Transaction Identifier ist das einzige Feld, das nicht direkt über eine Firewall getestet werden kann. In einer sauberen Implementierung sollte der Identifier für jede neue Anfrage um eins erhöht werden. Mit dieser Vorgabe ließe sich auch diese Möglichkeit eines verdeckten Kanals im Header schließen.

Payload

Da wir keinen Payload-Channel-Angriff umgesetzt haben, können wir auch keine konkrete Detektionsmaßnahme angeben. Grundsätzlich kann eine Untersuchung des Payloads nicht über eine Firewall funktionieren.Es muss mittels Deep-Packet-Inspection in den Paketinhalt geschaut werden und dieser auf Anomalien untersucht werden.

Artificial Retransmission

Da Modbus TCP/IP über TCP funktioniert und sich dieses um die erneute Zustellung verlorengegangener Pakete kümmert, kann eine Stateful-Firewall alle mehrfach gesendeten Modbus-Pakete filtern. Damit kann dieser Angriff unterbunden werden.

Interpacket Times

Von den umgesetzten Angriffen ist dies der am schwierigstem zu detektierende Angriff. Die Detektion ist zum Beispiel realisierbar, indem Zeitstempel der Modbus-Pakete auf Muster untersucht werden. Ohne Kenntnis des steganografischen Schlüssels ist dies nicht trivial [MWZ+16]. Die Kapazität des Kanals kann man durch die Verringerung des Timeout-Intervalls reduzieren.

Abbildung 5.1 zeigt die Kapazität der untersuchten Kanäle im Verhältnis zu den Kosten einer Detektions-/Vermeidungsstrategie.

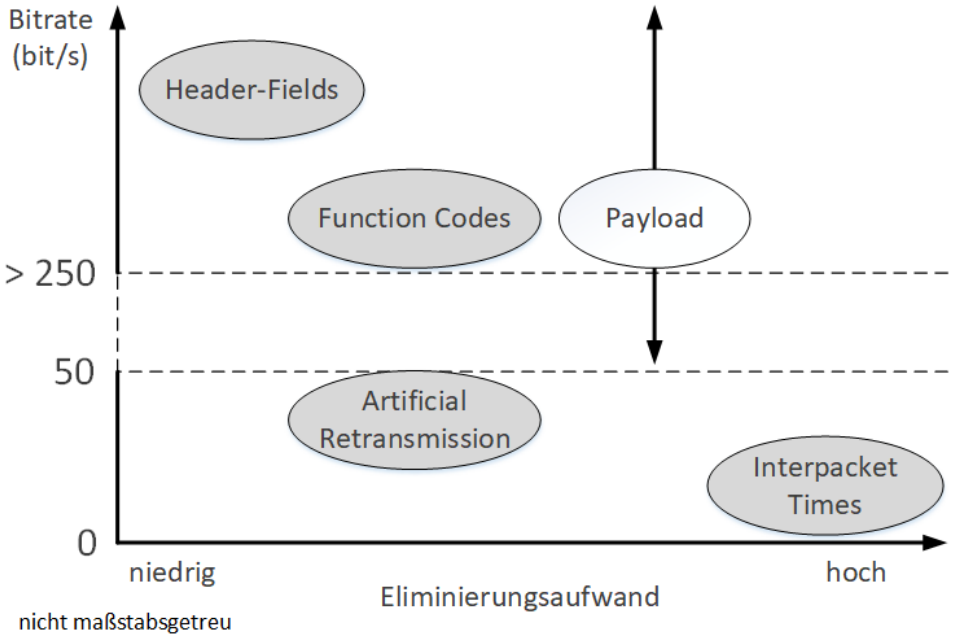

Abbildung 5.1: Einordnung der umgesetzten steganografischen Methoden bezüglich der erzielten Bitraten und dem Aufwand für die Eliminierung (Unterbindung) der Kommunikation

6 Fazit und Ausblick

(Uwe und Sophie)

In der vorliegenden Projektarbeit wurden vier steganografische Kanäle bezüglich des Modbus-Protokolls untersucht und eine Auswahl dieser implementiert. Diese vier Kanäle sind die beiden Speicherkanäle Modbus-Header und Payload, und die beiden Zeitkanäle Artificial Retransmission und Interpacket Time. Die Speicherkanäle wurden nur theoretisch bezüglich ihrer Kapazität und der Möglichkeit ihrer Umsetzung behandelt. Artificial Retransmission und Interpacket Time wurden implementiert und anhand eines Versuchsaufbaus getestet, welcher die quelloffene Software OpenPLC und ScadaBR verwendet. Die größten Schwierigkeiten bei der Umsetzung bereitete der Versuchsaufbau, weil die Software unzureichend dokumentiert war. Mithilfe der angepassten Implementierung von OpenPLC konnte die steganografische Kommunikation umgesetzt und zu Auswertungszwecken aufgezeichnet werden. Die Ergebnisse stellen die oberen Grenzen für die Bitraten vor, mit welchen die verdeckten Nachrichten im Versuchsaufbau übertragen wurden. Zuletzt wurden Detektions- und Vermeidungsmaßnahmen untersucht. Insbesondere kann der Einsatz von Stateless- und Stateful-Firewalls steganografische Kommunikation über das Modbus TCP/IP Protokoll unterbinden, wenn Artificial Retransmission und die Modifikation des MBAP-Headers ausgenutzt werden. Die Detektion von Angriffen, die die Payload und die Interpacket Time ausnutzen, erfordern dagegen aufwendige Gegenmaßnahmen, bei denen auf Applikations-Ebene systematische Untersuchungen vorgenommen und der Datenverkehr ausgewertet werden muss.

Da der Versuchsaufbau OpenPLC verwendet und dieses keine Implementierung von TLS enthält, ist es wünschenswert auch für diesen Anwendungsfall zu untersuchen, inwiefern die Verschlüsselung eine Auswirkung auf die Detektion und Vermeidung eines steganografischen Kanals hat.
Ausgehend von der Komplexität eines Versuchsaufbaus ist vorstellbar, dass auch spezielle Anwendungsfälle untersucht werden, die einen komplexeren Datenaustausch mithilfe von Registern beinhalten. In diesem Fall würden umfangreichere Payload-Inhalte eine Manipulation dieser erlauben, sodass der steganografische Speicherkanal bzgl. seiner Detektierbarkeit untersucht werden kann.
Auch ein Vergleich der steganografischen Kanäle mit anderen Modbus-Protokolltypen oder sogar anderen Feldbussystemen wäre denkbar.
Zuletzt beinhaltet die vorliegende Implementierung noch keine Fehlererkennung bzw. -korrektur für die zu übertragenden Daten. Diese könnte z.B. mithilfe von Cyclic Redundancy Check oder Hamming-Code realisiert werden.

Appendices

A. Ladder-Logic

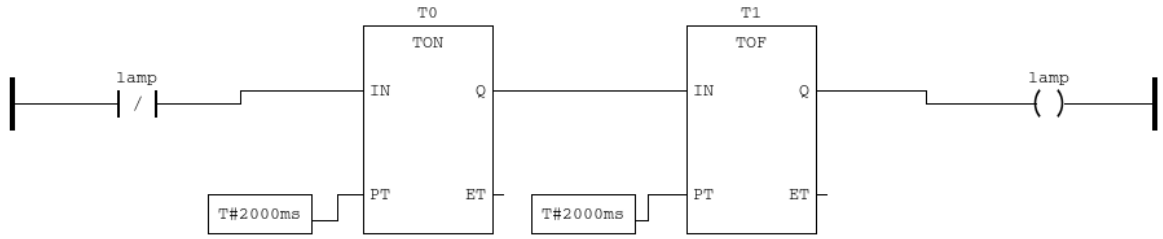

Abbildung .1: Ladder-Logik für oszillierende LED.

```
PROGRAM Oscilation
  VAR
    lamp AT %QX0.3 : BOOL;
  END_VAR
  VAR
    T1 : TOF;
    T0 : TON;
  END_VAR

  T0(IN := NOT(lamp), PT := T#2000ms);
  T1(IN := T0.Q, PT := T#2000ms);
  lamp := T1.Q;
END_PROGRAM


CONFIGURATION Config0

  RESOURCE Res0 ON PLC
    TASK task0(INTERVAL := T#20ms,PRIORITY := 0);
    PROGRAM instance0 WITH task0 : Oscilation;
  END_RESOURCE
END_CONFIGURATION
```

B. USAGE.txt

```
# Usage instructions
## Prerequisites
- OpenPLC and ScadaBR set up
- C++ compiler
```

```
- tshark installed on receiving node (machine running linux as an intermediate node to the connection anywhere between OpenPLC and ScadaBR)

## Installation
### On receiving node
- make receive.bash executable, e.g. "chmod +x receive.bash"
- compile codec.cpp and file2bitstream.cpp, e.g. "g++ -o codec codec.cpp" and "g++ -o file2bitstream file2bitstream.cpp"

### On OpenPLC
- make send.bash executable, e.g. "chmod +x receive.bash"
- compile codec.cpp and file2bitstream.cpp, e.g. "g++ -o codec codec.cpp" and "g++ -o file2bitstream file2bitstream.cpp"
- in OpenPLC, replace the following files with the modified versions provided in this folder
  -> <path_to_openplc>/webserver/core/server.cpp
  -> <path_to_openplc>/utils/libmodbus_src/src/modbus-tcp.c
- recompile OpenPLC from source through following the OpenPLC installation instructions

## Run
### On receiving node
- look up network interface name, e.g. "ip a"
- start receive.bash, e.g. "./receive.bash -i <name of interface> -o <path of file to write output to>"

### On OpenPLC
- start send.bash, e.g. "./send.bash" and type in a message to send
=> if successful, the sent message should appear as output on the receiving node
=> the output should also refer to a temporary package capture file that can be looked at for more insight

## Options overview
TODO

## Things to do
- better logging output
- error correction
- whole ascii, maybe unicode
- elaborate testing
```

C. send.bash

```
1  #!/bin/bash
2  # reads a string from input or file, uses a coder to convert it into a bitstream and writes this stream into a temp file for further processing
3  # also creates a temp file that contains the specified steganographic channel that the bitstream will be transported on
4
5  set -euo pipefail
6
7  # Variables
8  INPUT_FILE="-"
9  MESSAGE_FILE="/tmp/message.txt"
10 CODE="ascii"
11 STARTING_SEQUENCE="1111"
12 MESSAGE_SIZE_INFO="16"
13 STEGO_MODE="1"
14 STEGO_MODE_FILE="/tmp/stego_mode.txt"
15 MESSAGE=""
16 BITSTREAM=""
17 usage="$(basename "$0") [-i INPUT_FILE] [-c CODE] \
18 [-s STARTING_SEQUENCE]
19
20 where:
21   -i INPUT_FILE   path to input file containing a message to send. ff not Specified, the script will ask for input.
22   -c CODE     defaults to ascii code. specifies code used to convert to bit stream.
```

```
23    -s STARTING_SEQUENCE  defaults to 1111. specifies sequence signaling to the receiver ↩
          that a message is arriving.
24      -m STEGO_MODE           defaults to 1. specifies the steganographic channel ↩
              used for messaging. for more info on nodes, see the USAGE.txt file
25    -x MESSAGE_SIZE_INFO  defaults to 16. specifies the amount of bits that describe the ↩
          message size. this directly impacts the maximum message size."
26
27  # Options
28  while getopts ":i:o:c:s:m:" opt; do
29    case $opt in
30    i)  INPUT_FILE=$OPTARG
31        ;;
32    c)  CODE=$OPTARG
33        ;;
34    s)  STARTING_SEQUENCE=$OPTARG
35        ;;
36    m)  STEGO_MODE=$OPTARG
37              ;;
38    x)  MESSAGE_SIZE_INFO=$OPTARG
39              ;;
40    *)
41        echo "$usage" >&2
42        exit 1
43        ;;
44    esac
45  done
46
47  # Functions
48  ## checks whether message file already exists, ask to remove if yes
49  check() {
50    if [ -f $MESSAGE_FILE ]
51    then
52      echo "$MESSAGE_FILE already exists! Remove it? (y/n)[n]"
53      read REMOVE_FILE
54      if [ "$REMOVE_FILE" = "y" ]
55      then
56        rm $MESSAGE_FILE
57        if [ -f $MESSAGE_FILE ]
58        then
59          echo "Error removing file! Exiting..."
60          exit 1
61        fi
62        echo "Removed $MESSAGE_FILE"
63      else
64        exit 1
65      fi
66    fi
67    echo $STEGO_MODE > $STEGO_MODE_FILE
68  }
69
70  ## reads message from file or stdin
71  readInput() {
72    if [ ! -f $INPUT_FILE ]
73          then
74      echo "No (valid) input file specified. Please input message you want to convert to ↩
          bitstream and hit ENTER: "
75      read MESSAGE
76    else
77      echo "Reading content from $INPUT_FILE ..."
78      MESSAGE=$(<$INPUT_FILE)
79    fi
80  }
81
82  ## codes message to bitstream, appends starting sequence and size info
83  code() {
84    BITSTREAM="$(./codec code $CODE $STARTING_SEQUENCE $MESSAGE_SIZE_INFO $MESSAGE)"
85  }
86
87  ## writes bitstream to output file
88  writeOutput() {
89    echo "Message: $MESSAGE"
90    echo "Bitstream: $BITSTREAM"
91    echo "Writing bitstream to file ..."
```

```
  echo $BITSTREAM > $MESSAGE_FILE
}

## logs message and stego mode file paths
log() {
  echo "Written bitstream to $MESSAGE_FILE."
  echo "Written steganographic mode to $STEGO_MODE_FILE."
}

# starting point
check
readInput
code
writeOutput
log
echo "Done."
```

D. codec.cpp

Anmerkung: Aus Formatierungsgründen wurden alle Apostrophe aus dem Programmcode gelöscht.

```
#include <iostream>
#include <sys/types.h>
#include <sys/stat.h>
#include <map>
#include <string>
#include <bitset>

std::map<char, std::string> asciiMap = {{ , "00100000"}, {a ,"01100001"}, {b ,"01100010"}, {c ,"01100011"}, {d ,"01100100"}, {e ,"01100101"}, {f ,"01100110"}, {g ,"01100111"}, {h ,"01101000"}, {i ,"01101001"}, {j ,"01101010"}, {k ,"01101011"}, {l ,"01101100"}, {m ,"01101101"}, {n ,"01101110"}, {o ,"01101111"}, {p ,"01110000"}, {q ,"01110001"}, {r ,"01110010"}, {s ,"01110011"}, {t ,"01110100"}, {u ,"01110101"}, {v ,"01110110"}, {w ,"01110111"}, {x ,"01111000"}, {y ,"01111001"}, {z ,"01111010"}, {A ,"01000001"}, {B ,"01000010"}, {C ,"01000011"}, {D ,"01000100"}, {E ,"01000101"}, {F ,"01000110"}, {G ,"01000111"}, {H ,"01001000"}, {I ,"01001001"}, {J ,"01001010"}, {K ,"01001011"}, {L ,"01001100"}, {M ,"01001101"}, {N ,"01001110"}, {O ,"01001111"}, {P ,"01010000"}, {Q ,"01010001"}, {R ,"01010010"}, {S ,"01010011"}, {T ,"01010100"}, {U ,"01010101"}, {V ,"01010110"}, {W ,"01010111"}, {X ,"01011000"}, {Y ,"01011001"}, {Z ,"01011010"}};
std::map<std::string, char> asciiDecodeMap ={{"00100000",  }, {"01100001",a }, {"01100010",b }, {"01100011",c }, {"01100100",d }, {"01100101",e }, {"01100110",f }, {"01100111",g }, {"01101000",h }, {"01101001",i }, {"01101010",j }, {"01101011",k }, {"01101100",l }, {"01101101",m }, {"01101110",n }, {"01101111",o }, {"01110000",p }, {"01110001",q }, {"01110010",r }, {"01110011",s }, {"01110100",t }, {"01110101",u }, {"01110110",v }, {"01110111",w }, {"01111000",x }, {"01111001",y }, {"01111010",z }, {"01000001",A }, {"01000010",B }, {"01000011",C }, {"01000100",D }, {"01000101",E }, {"01000110",F }, {"01000111",G }, {"01001000",H }, {"01001001",I }, {"01001010",J }, {"01001011",K }, {"01001100",L }, {"01001101",M }, {"01001110",N }, {"01001111",O }, {"01010000",P }, {"01010001",Q }, {"01010010",R }, {"01010011",S }, {"01010100",T }, {"01010101",U }, {"01010110",V }, {"01010111",W }, {"01011000",X }, {"01011001",Y }, {"01011010",Z }};

/**
 * adds a header to the bitstream containing the message
 * the header consists of a user-specified starting sequence (default at this time is 1111), as well as a 16bit size info (size of message + header)
*/
void prepend_header(std::string& bitstream, std::string starting_sequence, std::string sizeInfo_length){
  size_t sizeInfo_length_converted = atoi(sizeInfo_length.c_str());
  // calculate size of packet
  size_t sizeInfo = starting_sequence.length() + sizeInfo_length_converted + bitstream.length();
  std::bitset<16> sizeInfo_bitstream(sizeInfo);
```

```
  bitstream = starting_sequence + sizeInfo_bitstream.to_string() + bitstream;
}

/**
 * uses header to recognize start and end of message
 * then removes the header from the bitstream so that only data bits are left
*/
void remove_header(std::string& data, std::string starting_sequence, std::string sizeInfo_length){
  size_t start_index;
  size_t starting_sequence_length = starting_sequence.length();
  bool sequence_found = false;
  // looks for starting sequence inside data and remember start_index if found
  for(size_t i = 0; i < data.length() - starting_sequence.length(); i++){
    sequence_found = true;
    for(size_t j = 0; j < starting_sequence.length(); j++){
      if(data[i+j] != starting_sequence[j]){
        sequence_found = false;
      }
    }
    if(sequence_found){
      start_index = i;
      break;
    }
  }
  // extract bits that define message size (size is defined by N=sizeInfo_length bits after starting sequence)
  std::string sizeInfo_bitstream;
  for(size_t i = start_index + starting_sequence.length(); i < start_index + starting_sequence.length() + atoi(sizeInfo_length.c_str()) && i < data.length(); i++){
    sizeInfo_bitstream += data[i];
  }
  // get sizeInfo from bits
  int sizeInfo = std::stoi(sizeInfo_bitstream, nullptr, 2);

  std::string message;
  for(size_t i = start_index + starting_sequence.length() + atoi(sizeInfo_length.c_str()); i < start_index + sizeInfo; i++){
    message += data[i];
  }
  data = message;
}

/**
 * checks whether a complete message has yet been received
*/
bool check_for_message(std::string& data, std::string starting_sequence, std::string sizeInfo_length){
  size_t start_index;
  size_t starting_sequence_length = starting_sequence.length();
  bool sequence_found = false;
  // looks for starting sequence inside data and remember start_index if found
  for(size_t i = 0; i < data.length() - starting_sequence.length(); i++){
    sequence_found = true;
    for(size_t j = 0; j < starting_sequence.length(); j++){
      if(data[i+j] != starting_sequence[j]){
        sequence_found = false;
      }
    }
    if(sequence_found){
      start_index = i;
      break;
    }
  }
  if(!sequence_found){
    return false;
  }
  // extract bits that define message size (size is defined by N=sizeInfo_length bits after starting sequence)
  std::string sizeInfo_bitstream;
  for(size_t i = start_index + starting_sequence.length(); i < start_index + starting_sequence.length() + atoi(sizeInfo_length.c_str()) && i < data.length(); 
```

```
      i++){
    sizeInfo_bitstream += data[i];
  }
  if(sizeInfo_bitstream.length() < atoi(sizeInfo_length.c_str())){
    return false;
  }
  // get sizeInfo from bits
  int sizeInfo = std::stoi(sizeInfo_bitstream, nullptr, 2);
  // compare data length - start_index vs sizeInfo
  return (data.length() - start_index >= sizeInfo);
}

/**
 * codes a string into a bitstream
 * incomplete asciicode is used; only letters and the space sign are recognized at this time
*/
std::string asciicode(std::string message){
  std::string bitstream;
  const char *message_char_array = message.c_str();
  for(size_t i = 0; message_char_array[i] != \0; i++) {
    bitstream+=asciiMap[message_char_array[i]];
  }
  return bitstream;
}

/**
 * decodes a bitstream into a string
 * incomplete asciicode is used; only letters and the space sign are recognized at this time
*/
std::string asciidecode(std::string bitstream){
  std::string message, tmp_string;
  const char *bitstream_char_array = bitstream.c_str();
  for(size_t i = 0; bitstream_char_array[i] != \0; i+=8) {
    tmp_string="";
    for(size_t j = i; j < i + 8; j++) {
      tmp_string+=bitstream_char_array[j];
    }
    message+=asciiDecodeMap[tmp_string];
  }
  return message;
}

/**
 * offers three actions:
 * - codes a string into a bitstream and adds a header containing a starting sequence and message size info
 * - checks, based on starting sequence and size info stored inside the header, whether a bitstream contains a complete message
 * - decodes a bitstream containing a complete message into a string (after extracting size info and removing the header)
 * - see send.bash and receive.bash for more info on how to call
*/
int main(int argc, char **argv){
  try{
    if(argc < 3 || argc > 6){
      throw "Bad number of arguments.";
    }
    std::string action = std::string(argv[1]);
    if(action == "code"){
      std::string code = std::string(argv[2]);
      std::string starting_sequence = std::string(argv[3]);
      std::string sizeInfo_length = std::string(argv[4]);
      std::string content = std::string(argv[5]);
      if(code == "ascii"){
        std::string bitstream = asciicode(content);
        prepend_header(bitstream, starting_sequence, sizeInfo_length);
        std::cout << bitstream;
      }
    }else if(action == "decode"){
      std::string code = std::string(argv[2]);
      std::string starting_sequence = std::string(argv[3]);
```

```
152         std::string sizeInfo_length = std::string(argv[4]);
153         std::string bitstream = std::string(argv[5]);
154         remove_header(bitstream, starting_sequence, sizeInfo_length);
155         if(code == "ascii"){
156           std::string message = asciidecode(bitstream);
157           std::cout << message;
158         }
159       }else if(action == "check"){
160         std::string starting_sequence = std::string(argv[2]);
161         std::string sizeInfo_length = std::string(argv[3]);
162         std::string bitstream = std::string(argv[4]);
163         std::cout << check_for_message(bitstream, starting_sequence, sizeInfo_length);
164       }
165     }catch(const std::exception& e){
166       std::cerr << e.what() << std::endl;
167     }catch(const char* e){
168       std::cerr << e << std::endl;
169     }
170     return 0;
171 }
```

E. server.cpp

```
1  //-----------------------------------------------------------------------------
2  // Copyright 2015 Thiago Alves
3  // This file is part of the OpenPLC Software Stack.
4  //
5  // OpenPLC is free software: you can redistribute it and/or modify
6  // it under the terms of the GNU General Public License as published by
7  // the Free Software Foundation, either version 3 of the License, or
8  // (at your option) any later version.
9  //
10 // OpenPLC is distributed in the hope that it will be useful,
11 // but WITHOUT ANY WARRANTY; without even the implied warranty of
12 // MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
13 // GNU General Public License for more details.
14 //
15 // You should have received a copy of the GNU General Public License
16 // along with OpenPLC.  If not, see <http://www.gnu.org/licenses/>.
17 //------
18 //
19 // This is the file for the network routines of the OpenPLC. It has procedures
20 // to create a socket, bind it and start network communication.
21 // Thiago Alves, Dec 2015
22 //-----------------------------------------------------------------------------
23
24 #include <stdio.h>
25 #include <stdlib.h>
26 #include <unistd.h>
27 #include <errno.h>
28 #include <netdb.h>
29 #include <string.h>
30 #include <pthread.h>
31 #include <fcntl.h>
32
33 /* for stego included libraries */
34 #include <fstream>
35 #include <sstream>
36 #include <algorithm>
37 #include <map>
38 #include <iostream>
39 #include <sys/types.h>
40 #include <sys/stat.h>
41 #include <string>
42
43 #include "ladder.h"
44
```

```
45 #define MAX_INPUT 16
46 #define MAX_OUTPUT 16
47 #define MAX_MODBUS 100
48
49
50 //------------------------------------------------------------------------------
51 // Verify if all errors were cleared on a socket
52 //------------------------------------------------------------------------------
53 int getSO_ERROR(int fd)
54 {
55    int err = 1;
56    socklen_t len = sizeof err;
57    if (-1 == getsockopt(fd, SOL_SOCKET, SO_ERROR, (char *)&err, &len))
58       perror("getSO_ERROR");
59    if (err)
60       errno = err;              // set errno to the socket SO_ERROR
61    return err;
62 }
63
64 //------------------------------------------------------------------------------
65 // Properly close a socket
66 //------------------------------------------------------------------------------
67 void closeSocket(int fd)
68 {
69    if (fd >= 0)
70    {
71       getSO_ERROR(fd); // first clear any errors, which can cause close to fail
72       if (shutdown(fd, SHUT_RDWR) < 0) // secondly, terminate the reliable delivery
73          if (errno != ENOTCONN && errno != EINVAL) // SGI causes EINVAL
74             perror("shutdown");
75       if (close(fd) < 0) // finally call close()
76          perror("close");
77    }
78 }
79
80 //------------------------------------------------------------------------------
81 // Set or Reset the O_NONBLOCK flag from sockets
82 //------------------------------------------------------------------------------
83 bool SetSocketBlockingEnabled(int fd, bool blocking)
84 {
85    if (fd < 0) return false;
86    int flags = fcntl(fd, F_GETFL, 0);
87    if (flags == -1) return false;
88    flags = blocking ? (flags & ~O_NONBLOCK) : (flags | O_NONBLOCK);
89    return (fcntl(fd, F_SETFL, flags) == 0) ? true : false;
90 }
91
92 //------------------------------------------------------------------------------
93 // Create the socket and bind it. Returns the file descriptor for the socket
94 // created.
95 //------------------------------------------------------------------------------
96 int createSocket(int port)
97 {
98     unsigned char log_msg[1000];
99     int socket_fd;
100    struct sockaddr_in server_addr;
101
102    //Create TCP Socket
103    socket_fd = socket(AF_INET,SOCK_STREAM,0);
104    if (socket_fd<0)
105    {
106        sprintf(log_msg, "Modbus Server: error creating stream socket => %s\n", ↩
               strerror(errno));
107        log(log_msg);
108        return -1;
109    }
110
111    //Set SO_REUSEADDR
112    int enable = 1;
113    if (setsockopt(socket_fd, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(int)) < 0)
114        perror("setsockopt(SO_REUSEADDR) failed");
115
116    SetSocketBlockingEnabled(socket_fd, false);
```

```
117
118     //Initialize Server Struct
119     bzero((char *) &server_addr, sizeof(server_addr));
120     server_addr.sin_family = AF_INET;
121     server_addr.sin_addr.s_addr = INADDR_ANY;
122     server_addr.sin_port = htons(port);
123
124     //Bind socket
125     if (bind(socket_fd,(struct sockaddr *)&server_addr,sizeof(server_addr)) < 0)
126     {
127         sprintf(log_msg, "Modbus Server: error binding socket => %s\n", strerror(errno)↩
                ;
128         log(log_msg);
129         return -1;
130     }
131
132     // we accept max 5 pending connections
133     listen(socket_fd,5);
134     sprintf(log_msg, "Modbus Server: Listening on port %d\n", port);
135     log(log_msg);
136
137     return socket_fd;
138 }
139
140 //-----------------------------------------------------------------------------
141 // Blocking call. Wait here for the client to connect. Returns the file
142 // descriptor to communicate with the client.
143 //-----------------------------------------------------------------------------
144 int waitForClient(int socket_fd)
145 {
146     unsigned char log_msg[1000];
147     int client_fd;
148     struct sockaddr_in client_addr;
149     socklen_t client_len;
150
151     sprintf(log_msg, "Modbus Server: waiting for new client...\n");
152     log(log_msg);
153
154     client_len = sizeof(client_addr);
155     while (run_modbus)
156     {
157         client_fd = accept(socket_fd, (struct sockaddr *)&client_addr, &client_len); //↩
              non-blocking call
158         if (client_fd > 0)
159         {
160             SetSocketBlockingEnabled(client_fd, true);
161             break;
162         }
163         sleepms(100);
164     }
165
166     return client_fd;
167 }
168
169 //-----------------------------------------------------------------------------
170 // Blocking call. Holds here until something is received from the client.
171 // Once the message is received, it is stored on the buffer and the function
172 // returns the number of bytes received.
173 //-----------------------------------------------------------------------------
174 int listenToClient(int client_fd, unsigned char *buffer)
175 {
176     bzero(buffer, 1024);
177     int n = read(client_fd, buffer, 1024);
178     return n;
179 }
180
181 //-----------------------------------------------------------------------------
182 // Process clients request
183 //-----------------------------------------------------------------------------
184 void processMessage(unsigned char *buffer, int bufferSize, int client_fd)
185 {
186     int messageSize = processModbusMessage(buffer, bufferSize);
187     write(client_fd, buffer, messageSize);
```

```
188 }
189
190 //--------------------------------------------------------------------------------
191 // Thread to handle requests for each connected client
192 //--------------------------------------------------------------------------------
193 void *handleConnections(void *arguments)
194 {
195     unsigned char log_msg[1000];
196     int client_fd = *(int *)arguments;
197     unsigned char buffer[1024];
198     int messageSize;
199
200     sprintf(log_msg, "Modbus Server: Thread created for client ID: %d\n", client_fd);
201     log(log_msg);
202
203     while(run_modbus)
204     {
205         //unsigned char buffer[1024];
206         //int messageSize;
207
208         messageSize = listenToClient(client_fd, buffer);
209         if (messageSize <= 0 || messageSize > 1024)
210         {
211             // something has  gone wrong or the client has closed connection
212             if (messageSize == 0)
213             {
214                 sprintf(log_msg, "Modbus Server: client ID: %d has closed the ↩
                        connection\n", client_fd);
215                 log(log_msg);
216             }
217             else
218             {
219                 sprintf(log_msg, "Modbus Server: Something is wrong with the  client ID↩
                        : %d message Size : %i\n", client_fd, messageSize);
220                 log(log_msg);
221             }
222             break;
223         }
224
225   /* modified part for steganographic mode 1 and 2 */
226         try{
227       const std::string filename="/tmp/stego_mode.txt";
228       const std::string filename2="/tmp/message.txt";
229             std::ifstream stego_mode_file(filename);
230             std::ifstream message_file(filename2);
231       if(stego_mode_file && message_file){
232               std::string stego_mode;
233               getline(stego_mode_file, stego_mode);
234               std::string bitstream;
235               getline(message_file, bitstream);
236               if(stego_mode.find("1") != std::string::npos){
237                 processMessage(buffer, messageSize, client_fd);
238                 if(bitstream[0] == "1"){
239                   processMessage(buffer, messageSize, client_fd);
240                 }
241               }else if(stego_mode.find("2") != std::string::npos){
242                 if(bitstream[0] == "0"){
243                   processMessage(buffer, messageSize, client_fd);
244                 }else if (bitstream[0] == "1"){
245                   usleep(150000);
246                   processMessage(buffer, messageSize, client_fd);
247                 }else{
248                   processMessage(buffer, messageSize, client_fd);
249                 }
250               }
251               const int result = remove("/tmp/message.txt");
252               std::ofstream new_message_file("/tmp/message.txt", std::ios_base::app);
253               for(size_t i = 1; i < bitstream.length(); i++){
254                 new_message_file << bitstream[i];
255               }
256               new_message_file.close();
257             }else{
258               processMessage(buffer, messageSize, client_fd);
```

```
259             }
260         }catch(const std::exception& e){
261            std::cout << e.what() << std::endl;
262         }
263     }
264 
265     //printf("Debug: Closing client socket and calling pthread_exit in server.cpp\n");
266     close(client_fd);
267     sprintf(log_msg, "Terminating Modbus connections thread\r\n");
268     log(log_msg);
269     pthread_exit(NULL);
270 }
271 
272 //---------------------------------------------------------------------------
273 // Function to start the server. It receives the port number as argument and
274 // creates an infinite loop to listen and parse the messages sent by the
275 // clients
276 //---------------------------------------------------------------------------
277 void startServer(int port)
278 {
279     unsigned char log_msg[1000];
280     int socket_fd, client_fd;
281 
282     socket_fd = createSocket(port);
283     mapUnusedIO();
284 
285     while(run_modbus)
286     {
287         client_fd = waitForClient(socket_fd); //block until a client connects
288         if (client_fd < 0)
289         {
290             sprintf(log_msg, "Modbus Server: Error accepting client!\n");
291             log(log_msg);
292         }
293 
294         else
295         {
296             int arguments[1];
297             pthread_t thread;
298             int ret = -1;
299             sprintf(log_msg, "Modbus Server: Client accepted! Creating thread for the ↩
                    new client ID: %d...\n", client_fd);
300             log(log_msg);
301             arguments[0] = client_fd;
302             ret = pthread_create(&thread, NULL, handleConnections, arguments);
303             if (ret==0)
304             {
305                 pthread_detach(thread);
306             }
307         }
308     }
309     close(socket_fd);
310     close(client_fd);
311     sprintf(log_msg, "Terminating Modbus thread\r\n");
312     log(log_msg);
313 }
```

F. file2bitstream.cpp

```
1 #include <fstream>
2 #include <sstream>
3 #include <algorithm>
4 #include <map>
5 #include <iostream>
6 #include <sys/types.h>
7 #include <sys/stat.h>
8 #include <map>
```

```
9  #include <string>
10 #include <bitset>
11
12 /**
13  * mode1 counts the amount of responses that are shown in the packet filter after a ↩
        respective query
14  * one response equals 0, two responses equals 1
15 */
16 std::string mode1(std::string pcap_path){
17   std::string bitstream;
18   size_t response_counter = 0;
19   try{
20     std::ifstream pcap_file(pcap_path);
21     if(!pcap_file){
22       throw std::ios::failure("Error opening file!");
23     }
24     std::string line;
25     while(getline(pcap_file,line)){
26       if(line.find("Modbus") != std::string::npos){
27         if(line.find("Query") != std::string::npos){
28           if(response_counter == 1){
29             bitstream += "0";
30           }else if(response_counter == 2){
31             bitstream += "1";
32           }
33           response_counter = 0;
34         }else if(line.find("Response") != std::string::npos){
35           response_counter++;
36         }
37       }
38     }
39     pcap_file.close();
40   }catch(const std::exception& e){
41     std::cout << e.what() << std::endl;
42   }
43   return bitstream;
44 }
45
46 /**
47  * mode2 calculates the packet inter arrival times and derives the bitstream from that.
48  * A reasonable threshold when differentiating between 0 and 1 corresponds to the time ↩
        delay used when implementing the steganographic message on the senders side.
49  * Also, both of the threshold here and the added delay before sending on the senders ↩
        side should be chosen based on the jitter within the concerned network.
50  * Here, inter arrival time <= 80ms equals 0 and inter arrival time > 80ms equals 1
51  * On the sender side, no added delay equals 0 and an added delay of 150ms equals 1
52 */
53 std::string mode2(std::string pcap_path, std::string pcap_arrival_times_path){
54   std::string bitstream;
55   try{
56     std::ifstream pcap_file(pcap_path);
57     std::ifstream file2(pcap_arrival_times_path);
58     if(!pcap_file){
59       throw std::ios::failure("Error opening file!");
60     }
61     std::string pcap_line, pcap_line_old;
62     std::string times_line, times_line_old;
63     std::ofstream outputfile ("debug.log"); //DEBUG
64
65     while(getline(pcap_file,pcap_line)){
66       getline(file2,times_line);
67       if(pcap_line.find("Modbus") != std::string::npos && pcap_line.find("Response") !=↩
             std::string::npos  &&
68          pcap_line_old.find("Modbus") != std::string::npos && pcap_line_old.find("↩
              Query") != std::string::npos){
69         float time = atof(times_line.c_str());
70         float time_old = atof(times_line_old.c_str());
71   float time_delta = time - time_old;
72         outputfile << times_line; //DEBUG
73         if(time_delta > 0.08){
74           bitstream += "1";
75         }else{
76           bitstream += "0";
```

```
        }
      }
      pcap_line_old = pcap_line;
      times_line_old = times_line;
    }
    outputfile.close(); //DEBUG
    pcap_file.close();
    file2.close();
  }catch(const std::exception& e){
    std::cout << e.what() << std::endl;
  }
  return bitstream;
}

/**
 * takes two (or three) arguments:
 * - the steganographic "mode", which chooses the appropriate function to extract a bitstream from the packet capture file
 * - the file path of said packet capture file
 * - in mode 2 (which uses packet inter-arrival times), as a third argument the file path to a respective list of the extracted arrival time from the packet capture
 * - returns the extracted bitstream to standard output, which can than be decoded into the received message
 * for more info, look into the USAGE.txt file as well as the receive.bash script
*/
int main(int argc, char **argv){
  try{
    if(argc < 3 || argc > 4){
      throw "Bad number of arguments.";
    }
    std::string stego_mode = std::string(argv[1]);
    std::string pcap_path = std::string(argv[2]);
    if(stego_mode == "1"){
      std::cout << mode1(pcap_path);
    }else if(stego_mode == "2"){
      std::string pcap_arrival_times_path = std::string(argv[3]);
      std::cout << mode2(pcap_path, pcap_arrival_times_path);
    }
  }catch(const std::exception& e){
    std::cerr << e.what() << std::endl;
  }catch(const char* e){
    std::cerr << e << std::endl;
  }
  return 0;
}
```

G. receive.bash

```
#!/bin/bash
# looks for modbus/tcp traffic on an interface, reads a message consisting of a bitstream from a specified steganographic channel and uses a decoder to convert it into a string

set -euo pipefail

# Variables
INTERFACE=""
OUTPUT_FILE=""
CODE="ascii"
STARTING_SEQUENCE="1111"
MESSAGE_SIZE_INFO="16"
STEGO_MODE="1"
MESSAGE=""
BITSTREAM=""
MESSAGE_COMPLETE="0"
TMP_PCAP_FILE=""
TMP_TXT_FILE=""
```

```
18 usage="$(basename "$0") [-i INTERFACE] [-o OUTPUT_FILE] [-c CODE] \
19 [-s STARTING_SEQUENCE]
20
21 where:
22   -i INTERFACE    network interface on which to listen for traffic.
23         -o OUTPUT_FILE          path to output file where the message will be written ↩
                to.
24         -c CODE                 defaults to Unicode. specifies code used to convert ↩
                from bit stream.
25         -s STARTING_SEQUENCE    defaults to 1111. specifies sequence signaling to the ↩
                receiver that a message is arriving.
26         -m STEGO_MODE   defaults to 1. specifies the steganographic channel used for ↩
                messaging."
27
28 # Options
29 while getopts ":i:o:c:s:m:" opt; do
30         case $opt in
31         i)  INTERFACE=$OPTARG
32             ;;
33   o)  OUTPUT_FILE=$OPTARG
34             ;;
35   c)  CODE=$OPTARG
36             ;;
37   s)  STARTING_SEQUENCE=$OPTARG
38             ;;
39   m)  STEGO_MODE=$OPTARG
40             ;;
41   *)
42             echo "$usage" >&2
43             exit 1
44             ;;
45   esac
46 done
47
48 # Functions
49 init() {
50   echo "Listening for message..."
51   TMP_TXT_FILE=""
52   TMP_PCAP_FILE=$(mktemp)
53   echo "Writing packet capture to $TMP_PCAP_FILE"
54   tshark -Y "mbtcp" -l -i $INTERFACE > $TMP_PCAP_FILE &
55   TSHARK_PID=$!
56 }
57
58 ## loop that checks the packet capture until one complete message has been received
59 loop() {
60   while [ "$MESSAGE_COMPLETE" != "1" ]
61   do
62     echo "Received Bitstream: $BITSTREAM"
63     if [ -f $TMP_TXT_FILE ]
64     then
65       rm -f $TMP_TXT_FILE
66     fi
67     TMP_TXT_FILE=$(mktemp)
68     cp $TMP_PCAP_FILE $TMP_TXT_FILE
69     extractBitStream
70     checkForMessage
71     sleep 1
72   done
73 }
74
75 ## prepares packet capture file for convert of steganographic message to bitstream
76 extractBitStream(){
77         # stego mode one means doubled replies are a 1, single replies a 0
78         if [ "$STEGO_MODE" = "1" ]
79         then
80     BITSTREAM="$(./file2bitstream $STEGO_MODE $TMP_TXT_FILE)"
81   fi
82         # stego mode one means doubled replies are a 1, single replies a 0
83         if [ "$STEGO_MODE" = "2" ]
84         then
85     TMP_TIME_INFO_FILE=$(mktemp)
86                 awk {print $2} $TMP_TXT_FILE > $TMP_TIME_INFO_FILE
```

```
87      BITSTREAM="$(./file2bitstream $STEGO_MODE $TMP_TXT_FILE $TMP_TIME_INFO_FILE)"
88    fi
89  }
90
91  ## checks whether the whole message has arrived yet
92  checkForMessage(){
93    MESSAGE_COMPLETE="$(./codec check $STARTING_SEQUENCE $MESSAGE_SIZE_INFO $BITSTREAM)"
94  }
95
96  ## decodes bitstream into message string
97  decode() {
98    echo "Message received! Decoding..."
99    MESSAGE="$(./codec decode $CODE $STARTING_SEQUENCE $MESSAGE_SIZE_INFO $BITSTREAM)"
100 }
101
102 ## writes received message to a temporary file
103 writeOutput() {
104   OUTPUT_FILE=$(mktemp)
105   echo "Writing message to output file ..."
106         echo $MESSAGE > $OUTPUT_FILE
107 }
108
109
110 cleanup() {
111   echo "Stopping tshark..."
112   kill $TSHARK_PID
113 }
114
115 ## writes logs to standard output
116 log() {
117   echo "Bitstream: $BITSTREAM"
118   echo "Message: $MESSAGE"
119   echo "Written message to $OUTPUT_FILE"
120   echo "Written packet capture to $TMP_PCAP_FILE"
121 }
122
123 # starting point
124 init
125 loop
126 decode
127 writeOutput
128 cleanup
129 log
130 echo "Done."
```

Literaturverzeichnis

[ACR05] ACROMAG INCORPORATED, 30765 South Wixom Road, P.O. BOX 437, Wixom, MI 48393-7037 U.S.A. *Introduction to Modbus TCP IP*, 2005. URL: https://www.acromag.com/sites/default/files/Acromag_Intro_ModbusTCP_765A.pdf.

[AMP] AMPLICON. *Process Control and Automation using Modbus Protocol*. URL: https://www.amplicon.com/docs/white-papers/MODBUS-in-Process-control.pdf.

[CCCK10] Abbas Cheddad, Joan Condell, Kevin Curran, and Paul Mc Kevitt. Digital image steganography: Survey and analysis of current methods. *Signal Processing*, 90(3):727 – 752, 2010. URL: http://www.sciencedirect.com/science/article/pii/S0165168409003648, doi:https://doi.org/10.1016/j.sigpro.2009.08.010.

[ML05] Steven J. Murdoch and Stephen Lewis. Embedding covert channels into tcp/ip. In Mauro Barni, Jordi Herrera-Joancomartí, Stefan Katzenbeisser, and Fernando Pérez-González, editors, *Information Hiding*, pages 247–261, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[MWZ+16] Wojciech Mazurczyk, Steffen Wendzel, Sebastian Zander, Amir Houmansadr, and Krzysztof Szczypiorski. *Information Hiding in Communication Networks - Fundamentals, Mechanisms, Applications, and Countermeasures*. John Wiley & Sons, New York, 2016.

[WZFH15] Steffen Wendzel, Sebastian Zander, Bernhard Fechner, and Christian Herdin. Pattern-based survey and categorization of network covert channel techniques. *ArXiv*, abs/1406.2901, 2015. doi:10.1145/2684195.