HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Institute for
Internet Technologies
and Applications

IPSec Diagnose Tool

Semesterarbeit

HSR—Hochschule für Technik Rapperswil
Institut für Internet-Technologien und -Anwendungen

Frühjahrssemester 2015

Autoren:	Jan Balmer, Theo Winter
Betreuender Dozent:	Prof. Dr. Andreas Steffen
Betreuer:	Tobias Brunner
Gegenleser:	Prof. Dr. Josef Joller
Experte:	Dr. Ralf Hauser
Industriepartner:	Open Systems AG

Debug

Report Generation Date: 6. Mai 2015

LaTex Version: 2014/05/01

This is pdfTeX, Version 3.14159265-2.6-1.40.15 (TeX Live 2014) kpathsea version 6.2.0

<TODO:> Debug Seite muss entfernt werden vor Release.

Abstract

Das IPSec Diagnose Tool ist eine Commandline-Applikation mit dem Ziel die Fehler-Diagnose bei IPSec VPN Verbindungen zu vereinfachen. Dabei wurden zwei Haupt Use Cases bearbeitet. Zum einen das konstante Detektieren von Paket-Verlusten durch ein passives Capturen von IPSec ESP Paketen. Zum anderen die Möglichkeit eine periodische Überprüfung der MTU innerhalb eines VPN Tunnels durchzuführen. Um die MTU festzustellen wird eine Menge von unterschiedlich grossen ICMP Paketen mit einem "Don't fragmentFlag durch den Tunnel geschickt. Die erfolgreichen Pakete generieren eine Antwort und so kann die korrekte MTU innert kürzester Zeit festgestellt werden. Das IPSec Diagnose Tool ist so designet dass es als Daemon konstant auf den Routern auf beiden Seiten eines Tunnels läuft. Um den Performance-Impact möglichst gering zu halten wurde bei der Entwicklung auf die Programmiersprache Go gesetzt. IPSec Diagnose Tool setzt für das Paket Capturen auf die Library libpcap sowie den von Google entwickelten Wrapper gopacket. IPSec Diagnose Tool wurde gem. den Anforderungen und im Auftrag der Open Systems AG entwickelt. Open Systems AG betreibt im Auftrag ihrer Kunden rund um die Uhr ein weltweites Netz von IPSec VPN Tunnels.

Management Summary

Ausgangslage

Die Firma Open Systems AG betreibt im Auftrag ihrer Kunden rund um die Uhr ein weltweites Netz von IPSec VPN Tunnels. Dabei steht eine hohe Qualität und Verfügbarkeit dieser Tunnels an erster Stelle. Die Open Systems AG betreibt deshalb ein Mission Control Center welches die VPN Verbindungen jederzeit überwacht. Doch nicht alle Eigenschaften dieser Verbindungen lassen sich auf einfache Art kontrollieren. So wurden zum Beispiel Paket-Verluste bisher nur periodisch überprüft. Auch eine Überprüfung der konfigurierten MTU fand bisher nur bei Problemfällen statt. Das vorliegende IPSec Diagnose Tool soll eine Möglichkeit bieten die Überwachung der IPSec Tunnels durch eine konsequente Überprüfung von Paket-Verlusten und einer periodischen Detektion der MTU zu vereinfachen.

Technologie

Das IPSec Diagnose Tool ist als Kommandozeilen Applikation realisiert welches 24/7 auf den Routern der Open Systems AG betrieben wird. Um einen möglichst kleinen Memory-Footprint zu hinterlassen und eine gute Performance zu bieten wurde entschieden Go als Programmiersprache einzusetzen. Als PCAP Library wird libpcap mit dem von Google entwickelten "gopacketWrapper eingesetzt. Um das IPSec Diagnose Tool zu konfigurieren wird jeweils ein JSON-Konfigurationsfile erstellt. Um zum Beispiel bei hohem Paket-Verlust eine Meldung abzusetzen wird Syslog verwendet.

Ergebnis

Als Ergebnis liegt eine leichtgewichtige, konfigurierbare und einfach zu bedienende Kommandozeilen Applikation vor. Die Applikation führt selbständig durch dass passive Abfangen von Paketen eine Paket-Verlust Kontrolle durch und meldet dann potentielle Probleme. Ausserdem wird via Cronjob, periodisch eine MTU-Bestimmung getriggert, die parallel für alle Konfigurierten Tunnels die MTU meldet.

Ausblick

Die vorliegende Applikation ist so aufgebaut dass Teile davon auch als Library verwendet werden können. Da wir das IPSec Diagnose Tool unter einer Open Source Lizenz veröffentlicht haben ist es denkbar dass Teile davon in anderen Go-Applikationen integriert werden. Oder aber das IPSec Diagnose Tool selbst wird durch weitere Diagnose-Möglichkeiten erweitert.

Erklärung der Eigenständigkeit

Hiermit erklären wir,

- dass wir die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt haben, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt sind oder was mit dem Betreuer schriftlich vereinbart wurde,
- dass wir sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben haben und,
- dass wir keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt haben.

Rapperswil-Jona, 6. Mai 2015

<TODO:> Scan Unterschrift Theo

Theo Winter

<TODO:> Scan Unterschrift Jan

Jan Balmer

Inhaltsverzeichnis

1. Anforderungen

Dieses Kapitel enthält die Beschreibung und Ziele unseres Projekts.

1.1. Einleitung

1.2. Ausgangslage

1.3. Use Cases

1.3.1. Aktoren & Stakeholder

Aktoren & Stakeholder	Interessen
IPSec-Tunnel Administrator	Möchte einen IPSec Tunnel betreiben der einen möglichst geringen Paket-Verlust aufweist.
VPN Benutzer	Möchte eine schnelle, stabile Verbindung benutzen.
System (IPSecDiagTool)	Möchte den Administrator bei Problemen informieren und stets die beste MTU vorschlagen können.

1.3.2. UC1: Bestimmen von Paketverlusten

Der Paketverlust soll passiv, durch auslesen der ESP-Sequenznummern, vom IPSec Diagnose Tool ermittelt werden. Der Paketverlust soll sowohl Prozentual als auch mit einer konkreten Zahl an verlorenen Paketen pro Zeiteinheit ausgewiesen werden. Die Zeiteinheit soll konfigurierbar sein.

1.3.3. UC1.1: Auslösen von Alerts bei überschrittenem Grenzwert

Wenn ein vorbestimmter Grenzwert beim Überprüfen des Paket-Verlust überschritten wird soll ein Alert ausgelöst werden. Dieser Alert kann in der Form eines Logfiles sein oder als Nachricht an einen Syslog-Server. Der Grenzwert soll konfigurierbar sein.

1.3.4. UC2: Bestimmen der optimalen MTU

Durch einen Tunnel werden testweise Pakete unterschiedlicher Grösse gesendet und auf der anderen Seite wieder aufgezeichnet. Dabei wird die ideale MTU ermittelt welche zu keiner Fragmentierung führt.

1.3.5. UC2.1: Periodische Bestimmung der idealen MTU

Alle Tunnels sollen periodisch, automatisch vom IPSec Diagnose Tool analysiert werden. Dabei wird die ideale MTU wie in UC1 beschrieben bestimmt. Dies kann mittels eines integrierten Daemon oder Cron-Jobs realisiert werden.

1.3.6. UC3: Konfiguration

Das IPSec Diagnose Tool soll grundsätzliche den Verkehr von allen Tunnels aufzeichnen. Es soll jedoch auch möglich sein anhand eines Konfigurationsfiles nur den Verkehr eines spezifischen Tunnels aufzuzeichnen. Dies wird durch das konfigurieren einer Source- und Destination Adresse erreicht.

1.3.7. Optional UC4: Passiv die IKE Header untersuchen

Das IPSec Diagnose Tool bietet die Möglichkeit passiv die IKE Header zu untersuchen und dabei Probleme festzustellen. Dieser Use Case wird nur bearbeitet wenn alle anderen Use Cases erfüllt sind und es müsste dann noch genauer festgelegt werden welche Probleme erkannt werden können.

1.4. Funktionale Anforderungen

todo

1.5. Nichtfunktionale Anforderungen

1.5.1. Angemessenheit

- Es sollen möglichst alle der genannten Anforderungen erfüllt werden.
- Die Applikation soll schlank und in einem einheitlichen Stil programmiert werden. Zudem sollen die Vorschriften der Codeformatierung von Go übernommen werden. Dadurch wird eine höhere Übersichtlichkeit und leichtere Einarbeitung in den Code gewährleistet.

Funktionalität

- Das Window für die Gültigkeit von ESP Paketen ist konfigurierbar. Die Konfiguration führt zu einer individuelleren Erkennung von Paketverlusten. So kann auch bestimmt werden ab welcher Abweichung verspätetes Paket als verloren angesehen wird.
- Das Diagnose Tool funktioniert mindestens bis zu einer Datenrate von 300Mbit/s. Wobei dieser Wert nur bis zu einer bestimmten Mindestgrösse von Paketen gewährleistet werden kann.
- Für die Überprüfung der MTU fungiert eine Seite als Client und die andere als Server, wobei die gleiche Software verwendet werden kann. Dadurch wird die Einrichtung der Überprüfung erleichtert, da nicht darauf geachtet werden muss welcher Softwareteil eingesetzt werden muss.
- Es ist konfigurierbar welches Netzwerkinterface bei der Überwachung verwendet werden soll.

Testbarkeit

- Die Businesslogik muss über alle Use Cases (Normal verhalten) mit automatischen Unit-Tests abgedeckt sein.
- Für die Test soll das "testing"Package von Go verwendet werden. Dies bietet die Vorteile einer automatisierten Ausführung und leichten Überprüfung von Testresultaten.

Fehlertoleranz

- Die Installation des Tools kann automatisch überprüft werden.
- Das System muss auch nach einem Neustart in einem definierten Zustand sein. Dies auch bei einem unvorhergesehenen Abbruch des Programms.
- Um eine Fehlerhafte Ausführung und damit verbundene Schäden zu vermeiden, wird vor dem Start überprüft ob die eingestellte Konfiguration gültig ist.

Änderbarkeit

- Daten und Reports werden in einem Standard gespeichert, welcher auch von anderen Systemen gelesen werden kann (CSV, XML).
- Der Code inklusive Kommentar sind auf Englisch um eine leichtere Anpassbarkeit zu erreichen.

Anforderungen an Umgebung

- Das Tool ist unter dem Betriebssystem Linux lauffähig, es werden jedoch root Rechte benötigt.

- Für die Umgebung braucht es die PCAP Library sowie die Go Programming Language.

- Das Tool soll Open-Source sein und der Quellcode daher für jeden zugänglich und anpassbar.

2. Analyse

In diesem Kapitel wird unsere Wahl der Programmiersprache und PCAP-Library erläutert.

2.1. Einführung

Gemäss der Aufgabenstellung dieser Semester-/Bachelorarbeit ist die Programmiersprache für das IPSec Diagnose Tool frei wählbar mit der einzigen Voraussetzung dass man eine PCAP-Library einbinden kann. In der Inception-Phase des Projekts ging es daher darum eine geeignete Sprache und Bibliothek zu wählen.

2.2. Java und JNetPcap

JNetPcap ist eine Java Library die einen Wrapper für Libpcap umsetzt. Es ist Opensource und bietet eine Echtzeit Dekodierung der aufgezeichneten Pakete. Es werden zahlreiche Protokolle unterstützt und es ist möglich diese auf einfache Weise durch eigene Definitionen zu ergänzen. Der Code Selbs ist eine Mischung aus Nativ- und Javacode und bietet so eine gute Performance.
Es ist eine ausführliche Dokumentation sowie zahlreiche Codebeispiele vorhanden und ist daher leicht zu handhaben und erlernbar. Durch die JavaVirtualMachine hat es einen etwas grösseren Arbeitsspeicher bedarf als andere Umsetzungen.

2.3. Golang und goPacket

2.3.1. Golang

Golang, auch Go genannt, ist eine eher junge Programmiersprache seit 2007 von Google Inc. entwickelt wurde. Golang hat einen C-ähnlichen Syntax, bietet aber viele Eigenschaften von modernen Programmiersprachen wie zum Beispiel Garbage Collection, Type-Safety, Dynamic-Typing, Closures und eine grosse Standard-Library. Im Oktober 2009 wurde die Golang der Öffentlichkeit als Open Source zur Verfügung gestellt.

2.3.2. Evaluation

Unseren ersten Kontakt mit Golang haben wir durch GoProbe von Open Systems gewonnen. GoProbe erlaubt leichtgewichtiges aggregieren von Paketen und deren effiziente Speicherung. Eine Abfrage der gespeicherten Paketen ist via Querying Flows möglich.

Die Installation von Golang und das Builden von goProbe waren etwas harzig. Es hat aber schlussendlich geklappt. Die Erkenntnisse wie man goProbe erfolgreich installieren kann sind im Anhang dokumentiert.

GoProbe besteht aus drei Modulen. Zum einen ein Modul das selbst goProbe heisst und zum aufzeichnen von Paketen verwendet wird. Die von goProbe aufgezeichneten Pakete werden dann mit goDB gespeichert. GoDB ist eine speziell für Netzwerk-Pakete entwickelte Datenbank. Die gespeicherten Pakete können dann mit goQuery wieder abgefragt werden. Ausserdem steht noch ein optionales Modul namens goConvert zur Verfügung. Die drei Module sind aber ein vollständiges Programm und nicht Bibliotheken die wir einfach so in unser IPSec Diagnose Tool einbinden können. Das von uns entwickelte IPSec Diagnose Tool würde GoProbe als Vorbild nehmen aber wahrscheinlich keine Dependencies darauf haben. In der momentanen Evaluierungs-Phase ist es aber ein gutes Versuchsobjekt für die Performance-Tests.

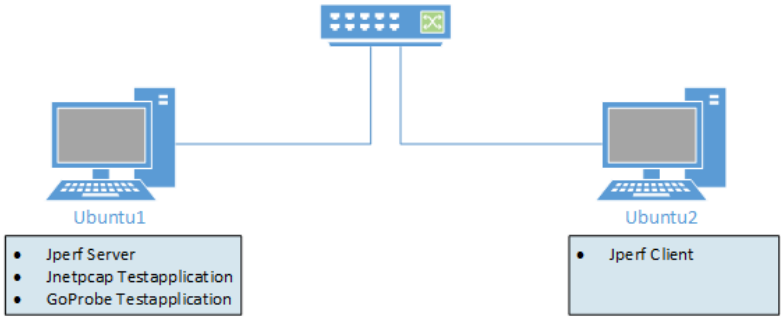
Golang selbst hat einige sehr angenehme Eigenschaften, so lässt sich der Code z.B. sehr einfach mit dem mitgelieferten godoc dokumentieren. Auch das Installieren von Dependencies via. 'go get package-name' ist sehr hilfreich, vorausgesetzt man hat die

GOPATH-Umgebungsvariablen korrekt gesetzt.

2.4. Performance Vergleich

2.4.1. Testaufbau

Zwei Desktop-Rechner der HSR, mit je 16GB Ram und Intel Xeon 3.4Ghz Quad-Core CPUs, sind via Gigabit-Lan miteinander verbunden. Auf den Rechnern läuft Ubuntu 14.04 x64 sowie jPerf und die jeweils getestete Software.



2.4.2. Testdurchführung

Auf einem der beiden Rechnern läuft jeweils jPerf im Server-Modus sowie die getestete Software. Auf dem anderen Computer läuft jPerf im Client-Modus. Via. jPerf wird nun soviel Traffic erzeugt um die 1Gbit/s-Leitung möglichst stark auszulasten, d.h. durchschnittlich 900mbit/s. Die getestete Software zeichnet dabei die ganzen Pakete auf und sollte dabei 300mbit/s an Traffic ertragen können. 300mbit/s sind gem. Open Systems AG die Lastspitzen mit denen etwa zu rechnen sind.

2.4.3. Ergebnisse

Sowohl mit JNetPcap auf Java als auch mit goProbe auf Golang lassen sich mehr als 300mbit/s an Verkehr aufzeichnen. GoProbe ist mit den Durchschnittlich 16% CPU Auslastung aber etwas performanter als JNetPcap. Die 31% CPU Lastspitze beim

Java Programm gibt es jeweils nur wenn das Programm zum ersten Mal gestartet wird und kommt daher dass dann zuerst eine JVM hochgefahren werden muss. Beim Speicherverbrauch hat goProbe aber deutlich die Nase vorne. So wird tatsächlich nur ein Bruchteil des physischen Speichers (RES) gegenüber Java verwendet.

Software	**CPU Top**	**CPU Ø**	**Mem Ø**	**VIRT[1]**	**RES[2]**	**SHR[3]**
JNetPcap	31%	20%	0.9%	7030296kb	147840kb	19588kb
goProbe	18%	16%	0.3%	315268kb	1964kb	1676kb

2.5. Entscheidung

Open Systems AG würde es bevorzugen wenn wir Golang statt Java einsetzen. Die Ergebnisse des Performance-Tests sprechen ebenfalls für Golang. Und wir haben durchaus auch das Interesse einmal eine neue Programmiersprache zu lernen. In Anbetracht dessen haben wir uns entschieden das IPSec Diagnose Tool mit Golang zu entwickeln.

[1]VIRT steht für die virtuelle Grösse eines Prozesses.
[2]RES steht für den tatsächlich, physisch verbrauchten Hauptspeicher.
[3]SHR zeigt wie viel von VIRT mit anderen Prozessen teilbar ist. Dazu gehören z.B. Shared Libraries.

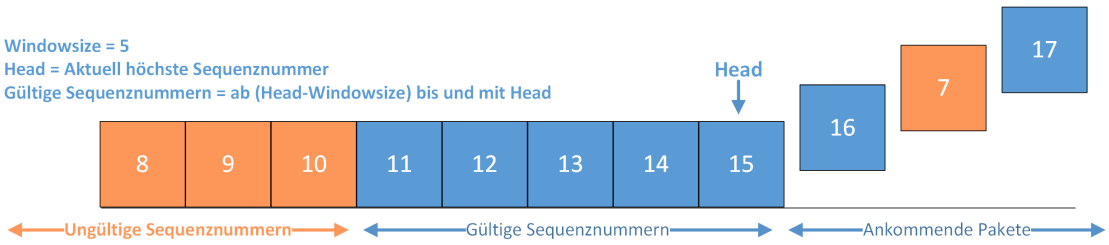
3. Implementation

3.1. Paketverlust erkennen

3.1.1. Einleitung

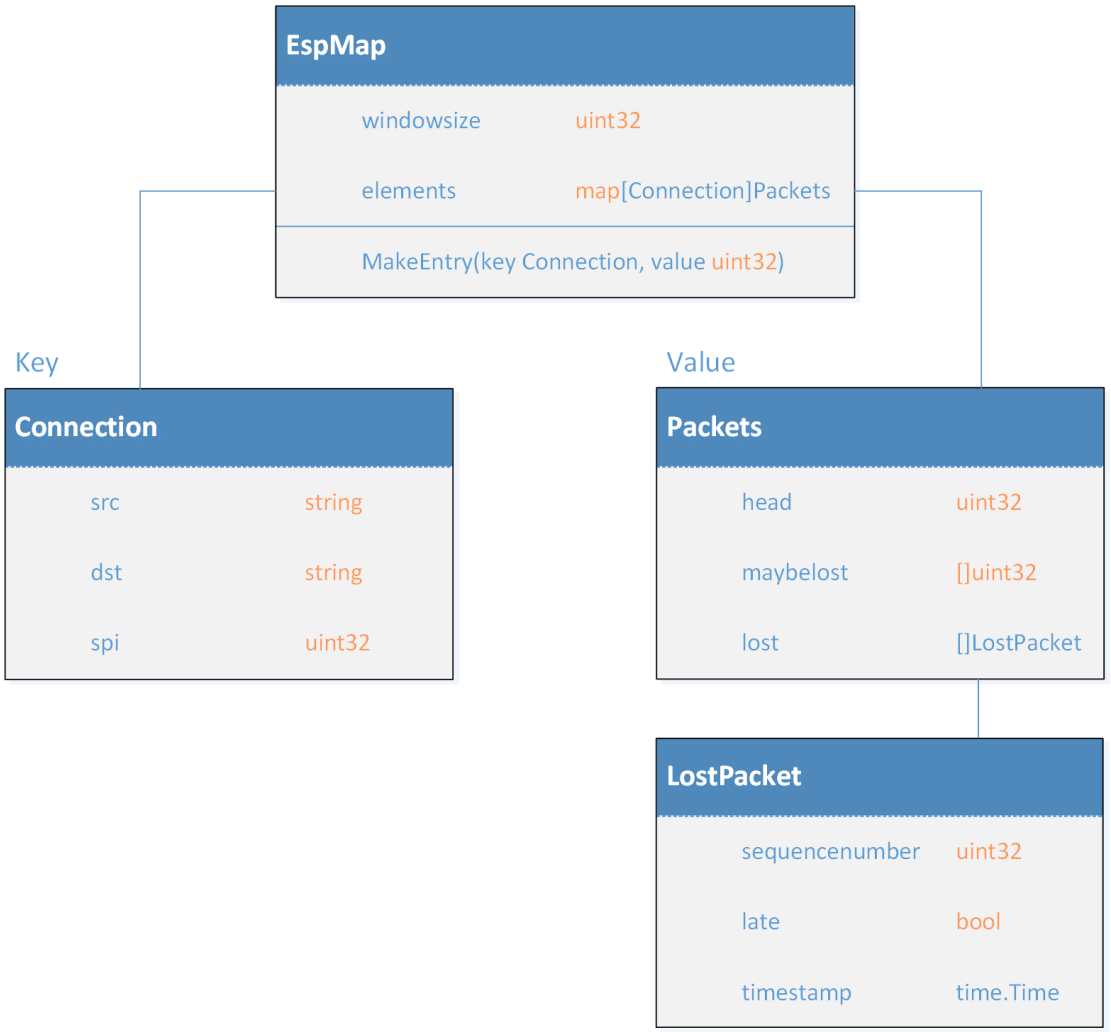
Falls ein Übertragungsmedium nicht korrekt funktioniert ist es möglich, dass Datenpakete nicht am Ziel ankommen oder mit einer zu grossen Verspätung. Da jedes ESPPaket mit einer Sequenznummer versehen ist, können diese Paketverluste erkannt und gemeldet werden.

Die ESPPakete Besitzen zum Schutz vor Replay-Angriffen eine Sequenznummer sowie einen bestimmten Gültigkeitsbereich (Windowsize).
Die Pakete können zum Beispiel durch Loadbalancing über unterschiedliche Pfade verschickt werden. Dadurch kann die Reihenfolge der Pakete am Ziel nicht mehr gewährleistet werden. Die zu spät ankommenden Pakete müssen daher überprüft werden, ob die Sequenznummer aktuell noch gültig ist (innerhalb des Window).



3.1.2. Datenstruktur für die ESPVerbindungen

Da pro Verbindung separate Sequenznummern verwendet werden, werden die verschiedenen Verbindungen durch eine Hashmap verwaltet. Als Key wird eine Struktur aus Source, Destination und SPI verwendet. Als Value werden grundsätzlich zwei Listen für die Lost und MaybeLost Pakete benötigt. Zusätzlich wird noch ein Wert für den aktuellen Head gespeichert.



3.1.3. Erkennung des Paketverlusts

Die Paketverluste werden mit folgendem Algorithmus festgestellt. Vereinfachte Darstellung des Algorithmus in Pseudocode:

Neues Packet Verarbeiten

```
1  if(neuesPacket > Head){
2     if(neuesPacket != Head + 1){
3        Packete von Head bis neuesPacket in
4        MaybeLost speichern
5     }
6     Head = neuesPacket
7     CheckLost()
8  }else{
9     if(Head-WindowSize < neuesPacket){
10       Packet aus MaybeLost entfernen
11    }else{
12       Flag fuer zu spaet angekommen setzen
13    }
14 }
```

CheckLost

```
1  for(MaybeLost){
2     if(Head-WindowSize >= MaybeLostEintrag){
3        Packet in Lost Speichern
4        Packet aus MaybeLost entfernen
5     }
6  }
```

3.2. MTU Discovery

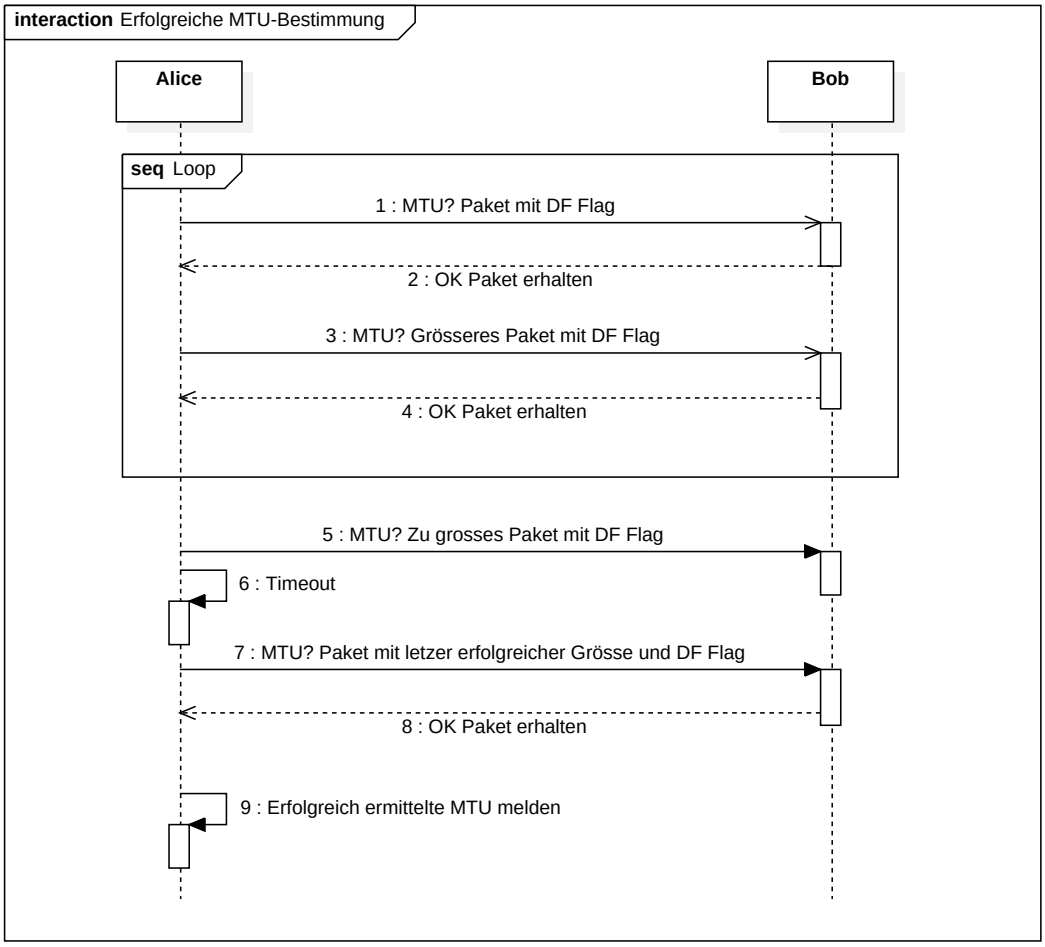
3.2.1. Einleitung

Mithilfe der MTU (Maximum Transmission Unit) wird festgelegt wie gross ein Netzwerkpaket maximal sein darf bevor es in mehrere Pakete aufgeteilt werden muss. Das finden der MTU läuft normalerweise via PMTUD (Path MTU Discovery) ab, dabei werden ICMP Pakete unterschiedlicher Grösse über die Verbindung gesandt und so festgestellt welches die maximale Grösse ist die noch ankommt. Das Problem dabei ist jedoch das ICMP Pakete in manchen Fällen von zu scharf konfigurierten Firewalls blockiert werden. Das heisst es kann vorkommen dass die MTU einer Verbindung nicht richtig konfiguriert ist und so kann man die Leitung nicht richtig ausnutzen. Um dieses Problem zu lösen haben wir eine MTU Discovery Funktion im IPSec Diagnose Tool entwickelt. Die Funktion verwendet ein ähnliches Vorgehen wie PMTUD, verlässt sich dabei aber nicht auf ICMP Pakete. Das IPSec Diagnose Tool läuft auf beiden Seiten eines IPSec Tunnels und überwacht jeweils passiv die Verbindung. Zu in der Konfiguration festgelegten Zeitpunkten versucht jeweils eine Seite die MTU zu ermitteln. Dabei werden laufend grössere Pakete übertragen bei denen ein "Do not fragmentFlag gesetzt wurde. So lange die Pakete auf der anderen Seite ankommen wird eine Antwort zurück gesendet. Falls keine Antwort zurück kommt weiss der Sender dass die Paketgrösse zu gross war und versucht das erneute versenden mit einer kleineren Paketgrösse. Am Schluss wird die erfolgreich festgestellte MTU protokolliert und die Timer beider Seiten wieder hochgestellt.

3.2.2. MTU-Bestimmung: Erfolgreich

Dieses Sequenz-Diagramm stellt den idealen Ablauf bei der MTU-Bestimmung dar. Alice schickt Bob ein Paket mit dem Kommando 'MTU?'. Dieses Paket hat eine typische Grösse gem. Erfahrungswerten. Momentan starten wir mit 500Bytes. Bob erhält das Paket und schickt eine Antwort mit dem Kommando 'OK'. Darauf erhöht Alice die Grösse des Pakets um einen konfigurierbaren Inkrementationswert und schickt es wieder zurück an Bob. Dieser Ablauf wird so lange wiederholt bis das Paket nicht mehr bei Bob ankommt. Weil das Paket nicht ankommt kann Bob keine Antwort senden und Alice läuft in ein Timeout. Alice nimmt jetzt den letzten erfolgreichen MTU-Wert und schickt Bob erneut ein Paket. Wenn das Paket erfolgreich ankommt, d.h. Bob ein 'OK' zurückschickt wurde

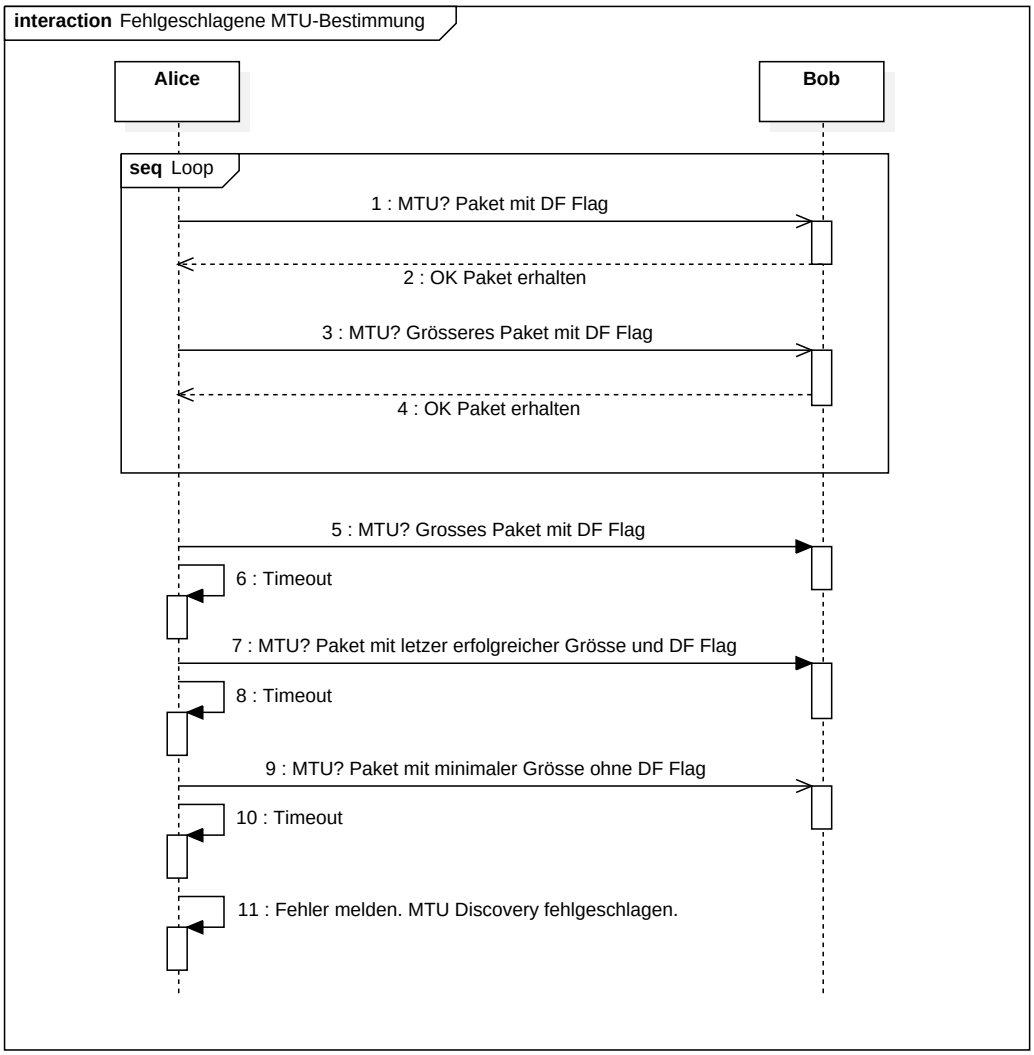
die MTU erfolgreich gefunden. Alice setzt nun einen Meldung ab das die ideal MTU für die Verbindung zwischen Alice und 2 gefunden wurde.



3.2.3. MTU-Bestimmung: Fehlerfall

Dieses Sequenz-Diagramm stellt einen möglichen Fehlerfall bei der Bestimmung der MTU dar. Es zeigt wie das IPSec Diagnose Tool den Fehlerfall behandelt. Alice versucht wie beim erfolgreichen Fall durch das versenden von immer grösseren Paketen die ideale MTU für die Verbindung mit Bob zu finden. Bei Punkt 5 sendet Bob keine Antwort mehr. Alice sendet Bob daher ein Paket mit der letzten erfolgreichen Grösse. Auch auf dieses Paket erhält Alice keine Antwort. ..todo Grafik anpassen.. Schlussendliche

meldet Alice die letzte erfolgreiche MTU mit einer Fehler-Nachricht und bricht den MTU Bestimmungsvorgang ab.



4. Projekt Management

4.1. Einführung

4.1.1. Zweck

Dieses Dokument dient zur Planung der Semester/Bachelorarbeit "IPSec Diagnose Tool ". Hier werden die Aufgabenstellung, die Iterationsplanung und die Meilensteine definiert.

4.1.2. Aufgabenstellung

Die Firma Open Systems betreibt im Auftrag ihrer Kunden rund um die Uhr ein weltweites Netz von VPN Verbindungen. Dabei steht eine hohe Qualität und Verfügbarkeit der IPSecTunnels an erster Stelle. Unter dem Linux-Betriebssystem soll ein Diagnose-Tool für IPSecVerbindungen erstellt werden das folgende Fähigkeiten hat:

Die Programmier- oder Skriptsprache, mit der das Tool erstellt wird ist offen. Es muss aber die Möglichkeit bestehen, die pcapLibrary zum passiven Aufzeichnen der Netzwerkpakete einzubinden.

- Passives Bestimmen der IPSecPaketverluste durch Erfassen der laufenden ESP Sequenznummern von ankommenden IPSecPaketen.

- Aktive Diagnose von IPsec Fragmentierungsproblemen und Bestimmung der optimalen MTU. Durch einen Tunnel werden IP Pakete variabler Grösse z.B. in der Form von IPMP Requests gesendet und überprüft, ob IP Fragmentierung auftritt und Fragmente verloren gehen.

4.1.3. Ziele

- Auswahl einer geigneten Programmiersprache und Wrappers für die pcapLibrary.
- Einarbeiten in IPSecTunnels und aufsetzen einer geeigneten Testumgebung.
- Spezifikation, Implementation und Test der IPSec Diagnose Tool Funktionalität.

4.1.4. Abgabetermine

Der offizielle Abgabetermin für die Semesterarbeit ist der Freitag, 29. Mai 2015. Der späteste Abgabetermin für die Bachelorarbeit ist der Freitag, 12. Juni 2015. Da diese Arbeit eine Kombination von Bachelor- und Semesterarbeit ist werden beide Arbeiten am 12. Juni 2015 mit einem jeweils passenden Titelblatt eingereicht.

4.2. Projektorganisation

4.2.1. Team

Das Projekt wird von Jan Balmer und Theo Winter bearbeitet. Die Projektleitung wird im Team abgewickelt.

4.2.2. Auftraggeber

Der Auftraggeber des Projekts ist die Open Systems AG. Der Kontakt zur Open Systems AG erfolgt über Stefan Keller und James Hulka. Betreut wird die Arbeit von Prof. Dr. Andreas Steffen und Tobias Brunner.

4.2.3. Besprechungen

Sitzungen finden jeweils wöchentlich am Freitag um 13:00 zwischen den Betreuern und dem Projektteam statt. Dabei wird typischerweise der aktuelle Projektstatus und das weitere Vorgehen besprochen.

Meetings mit den Auftraggebern finden drei Mal während dem Projektablauf statt. Einmal das Kickoff Meeting zu Beginn des Projekts, dann in der Mitte des Projekts zum zeigen des aktuellen Prototypen und zuletzt noch am Ende des Projekts in Form einer Präsentation.

4.2.4. Arbeitsumfang

Der Arbeitsaufwand für die Semesterarbeit wird mit total 240h angegeben. Was auf ca. 17.14h pro Woche heruntergerechnet werden kann. Bei der Bachelorarbeit geht man von einem Arbeitsaufwand von 20h pro Woche aus, sowie einer Fokus-Phase von 45h/Woche am Schluss der Arbeit. Total sollen für die Bachelorarbeit 370h geleistet werden.

Kombiniert sprechen wir somit von einem Arbeitsumfang von ca. 610h bei dieser Arbeit.

4.3. Arbeitsumgebung

4.3.1. Infrastruktur

Jedes Teammitglied hat ein persönliches Notebook welches hauptsächlich zur Projektbewältigung eingesetzt wird. Zusätzlich wurden von der HSR 2 Desktop-Rechner sowie ein VPS in der DMZ zur Verfügung gestellt. Zur Versionsverwaltung wird Git auf Github verwendet. Für die Zeiterfassung und Issue-Verwaltung wurde YouTrack auf dem VPS installiert.

4.3.2. Tools & Services

In diesem Projekt werden die folgende Tools & Services eingesetzt:

- **git**
  wird als Versionsverwaltungssystem eingesetzt.

- **Github.com**
  hostet unsere Git-Repositories, welche öffentlich zugänglich sind.

- **Jetbrains YouTrack**
  wird als Issue-Tool und zur Zeiterfassung eingesetzt.

- **Dropbox**
  wird fürs Backup der YouTrack Datenbank und als Datenablage für flüchtige Dokumente & Dateien verwendet.

- **LaTeX**
  wird als Programmiersprache für die Dokumentation eingesetzt.

- **Jenkins**
  wird als Continuous Integration Server zum automatischen generieren der LaTex Dokumentation verwendet.

- **Golang**
  wird als Programmiersprache für das IPSec Diagnose Tool eingesetzt.

- **goPacket**
  wir als PCAP-Library für das Projekt eingesetzt.

4.4. Qualitätsmassnahmen

4.4.1. Issue Verwaltung

Für die Issue Verwaltung und Zeiterfassung wird YouTrack von Jetbrains verwendet. Es erlaubt Issues zu erfassen, diese gewissen Meilensteinen zuzuordnen und auch jeweils die entsprechende Arbeitszeit zu verbuchen. Ausserdem können automatische Reports wie zum Beispiel ein "Burn-Down-Chart"generiert werden. So hat man den Projektablauf jeder Zeit im Blick. Wir haben YouTrack so konfiguriert dass es nahtlos mit unseren GitHub-Repositories zusammenarbeitet. So können wir z.B. mit einem Commit den Status von einem Issue direkt anpassen.

4.4.2. Versionskontrolle

Für dieses Projekt wurde eine Organisations ÏPSecDiagTooläuf Github.com erstellt. Diese Organisation unterhält 4 Repositories. Zwei persönliche Sandbox-Repositories zum Experimentieren und evaluieren der verschiedenen Pcap-Bibliotheken. Ein Haupt-Repository für das IPSec Diagnose Tool . In diesem Repository wird stets nur vollständig funktionsfähiger Code eingecheckt der dann mit einer sinnvollen Commit-Nachricht dokumentiert wird. Separate Features werden in diesem Repository aber in unterschiedlichen Branches entwickelt. Im vierten Repository befindet sich diese Dokumentation als LaTex Source-Dateien. Auch in dieses Repository wird nur Dokumentations-Code eingepflegt der kompiliert und durch eine verständliche Commit-Nachricht dokumentiert ist.

4.4.3. Dokumentation

Die Dokumentation wird während der Arbeit geschrieben und nicht in eine separaten Phase am Ende. So wird sichergestellt das unser gesamtes KnowHow erhalten bleibt. Es werden regelmässige Dokumentations-Reviews geplant um die Qualität der Dokumentation sicher zu stellen.

Ausserdem werden wir den Code-Dokumentation-Richtlinien von Google zur Verwendung mit Golang folgen. Dies hat den Vorteil dass via godoc zu jeder Zeit, automatisiert eine

komplette Dokumentation des Codes erzeugt werden kann. Zudem ist die Dokumentation im Code integriert so dass auch beim direkten lesen des Codes unsere Gedanken-Vorgänge anschaulich sind.

```
1  //LiveCapture captures all tcp & port 80 packets on eth0.
2  func LiveCapture() {
3     if handle, err := pcap.OpenLive("eth0", 1600, true, 0);
      err != nil {
4        panic(err)
5     } else if err := handle.SetBPFFilter("tcp and port 80");
      err != nil {
6        panic(err)
7     } else {
8        packetSource := gopacket.NewPacketSource(handle,
      handle.LinkType())
9        for packet := range packetSource.Packets() {
10          //Handling packets here
11          fmt.Println(packet.Dump())
12       }
13    }
14 }
```

4.4.4. Code Richtlinien

Grundsätzlich soll der Code gut lesbar sein, mit sauberen und aussagekräftigen Variablen-, Methoden-, und Klassennamen. Alle Kommentare im Code sind auf Englisch zu verfassen und sollen gem. den Google Dokumentation-Richtlinien erstellt werden (siehe oben). Änderungen am Code sollen regelmässig in unser Repository auf GitHub committed werden so dass man sie besser nachvollziehen kann.

Da Golang für uns eine neue Sprache ist und wir uns zum Zeitpunkt dieser Textstelle damit noch nicht gut auskennen gelten überall wo möglich die von Google Inc. verwendeten Code-Richtlinien.

4.4.5. Tests

Zu jeder erstellten Funktion sollen passenden Unit-Tests geschrieben werden um alle wichtigen Verhaltensweisen zu überprüfen. Die Unit-Tests werden gem. den Golang Vorlagen im selben Ordner wie der zu-testende Code gespeichert. Und zwar nach dem nachfolgenden Schema.

```
1 bin/
2 pkg/
3 src/
4     hsr.ipsecdiagtool/
5       main/
6             main.go                # command source
7       capture/
8             capture.go                # command source
9             capture_test.go           # test source
```

4.5. Iterationen und Meilensteine

4.5.1. Iterationsplanung

Zur Bewältigung dieser Arbeit verwenden wir einen angepassten Rational Unified Process Ablauf. RUP definiert 4 Phasen, welche iterativ mehrfach durchlaufen werden.

- **Inception**
  In der Inception Phase geht es darum die Details des Arbeitsauftrags auszuarbeiten und ein Fundament für das Projekt zu legen. Dies beinhaltet das Aufsetzen der Projekt-Management Software, erstellen von Repositories & File-Share. Aber auch das wählen des Vorgehensmodells.

- **Elaboration 1**
  In der ersten Elaboration Phase geht es darum sich für eine Programmiersprache

und einen Pcap-Wrapper zu entscheiden. Ausserdem wird der Projektplan erstellt und die Dokumentation aufgesetzt.

- **Elaboration 2**
  In der zweiten Elaboration Phase erstellen wir das Domain Modell sowie einen groben Prototypen unserer Applikation.

- **Construction 1**
  In der ersten Construction Phase soll die Hälfte der MUSS-Funktionalität implementiert sein. Dies beinhaltet Use Case 1 und Use Case 2.

- **Construction 2**
  In der zweiten Construction Phase soll die zweite Hälfte der MUSS-Funktionalität implementiert sein. Dies beinhaltet Sub-Use Case 1 und Sub-Use Case 2.

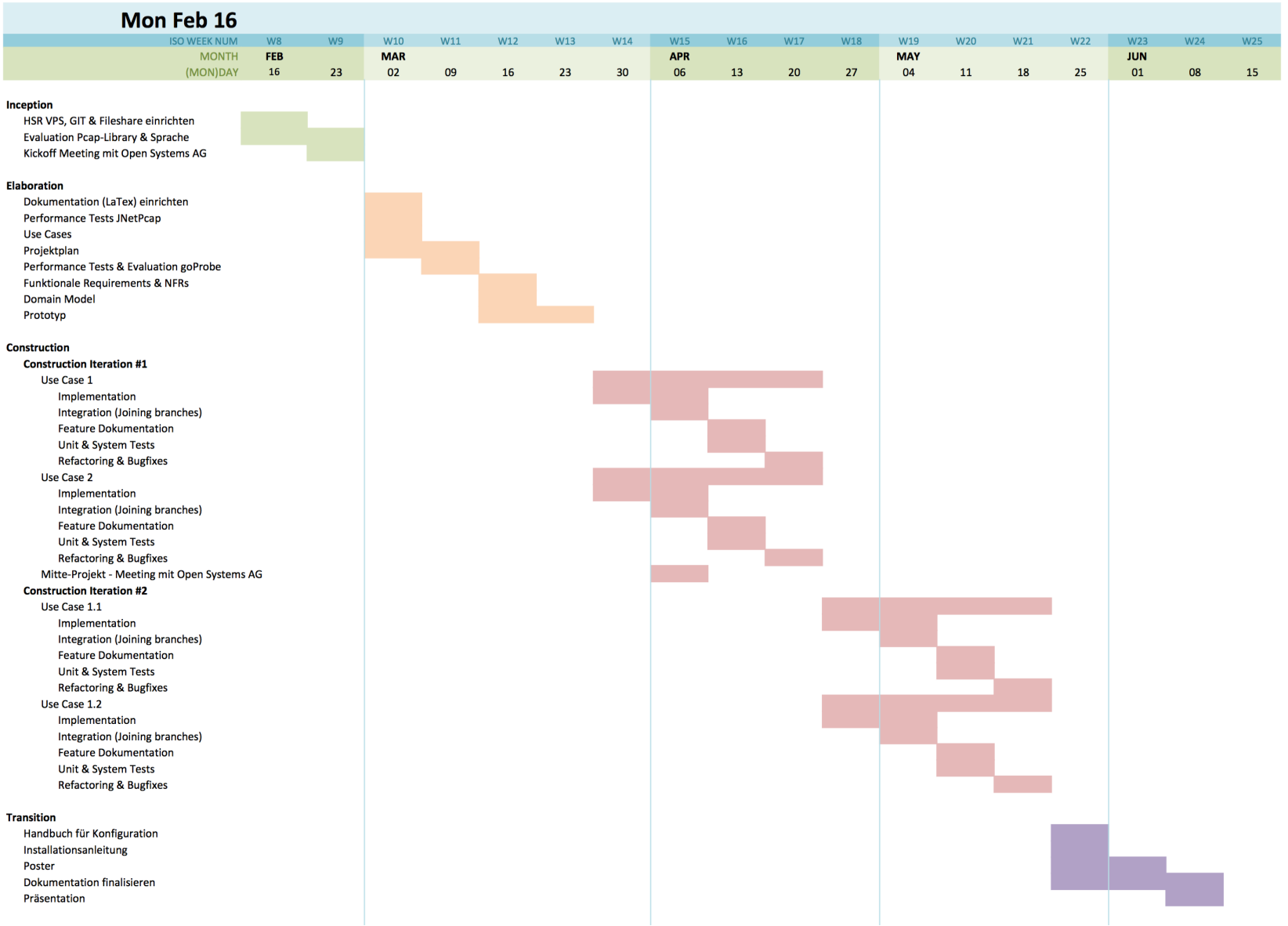
- **Transition**
  In der Transition-Phase sollen noch allfällige Bugs beseitigt werden. Sowie das Usermanual geschrieben und die Präsentation vorbereitet werden. Auch allfällige Rest-Dokumentationsarbeit soll in dieser Phase erledigt werden.

4.5.2. Meilensteine

Für das Projekt IPSec Diagnose Tool wurden die folgenden Meilensteine festgelegt:

MS	Name	Resultat	Datum
MS1	Projektplan	Projektplan erstellt und Iterationen geplant.	KW12
MS2	Anforderungspezifikation	Abgeschlossene Anforderungsspezifikationen, Use Cases und ein geplantes Domain Modell für den Prototypen.	KW13
MS3	Erster Prototyp	Ein erster Prototyp als Tech-Demo für das Mitte-Projekt-Meeting mit Open Systems.	KW15
MS4	Ende Construction #1	Basis Use Cases 1 und 2 implementiert.	KW17
MS5	Ende Construction #2	Erweiterte Use Cases 1.1 und 2.1 implementiert.	KW22
MS6	Abgabe SA/BA	Applikation ist fertig für den Release, Dokumentation wurd e abgegeben.	KW24

4.5.3. Zeitplan



4.6. Risikomanagment

4.6.1. Projektspezifisch

R1: Performance der PCAP Library reicht nicht	
Beschreibung	Die PCAP Library ist nicht genügend performant um alle Pakete aufzuzeichnen, es kommt zu Paket-Verlusten die nichts mit der Verbindung zu tun haben.
Massnahme	Wir führen bereits vor der definitiven Wahl der Library Performance Tests durch und können so feststellen ob sie unseren Ansprüchen (300mbit/s peak) genügt.
Vorgehen beim Eintreffen	Eine andere PCAP Library wählen.

R2: Java kann nicht verwendet werden	
Beschreibung	Die gewählt Programmiersprache ist nicht genügend performant, siehe Risiko 1, oder sie darf aus sonstigen Gründen nicht verwendet werden.
Massnahme	Die Wahl der Programmiersprache wird mit dem Kunden möglichst früh besprochen. Mögliche Performance-Probleme werden wie bei Risiko 1 durch einen Performance Test abgeklärt.
Vorgehen beim Eintreffen	Eine andere Programmiersprache und eine andere PCAP Library/Wrapper wählen.

4.6.2. Allgemeine

todo

4.6.3. Risikoschätzung

todo

Abkürzungsverzeichnis

CI	Continuous Integration
JVM	Java Virtual Machine
HSR	Hochschule für Technik Rapperswil
VPS	Virtual Private Server
IPSec	Internet Protocol Security
IP	Internet Protocol
RUP	Rational Unified Process
PCAP	Packet-Capture
MTU	Maximum Transmission Unit
PMTUD	Path MTU Discovery
ICMP	Internet Control Message Protocol
ESP	Encapsulating Security Payload
SPI	Security Parameters Index
VPN	Virtual Private Network

A. Usermanual

TODO: Usermanual ist in Englisch gem. Sitzung 2. This chapter explains how to use our IPSec Diagnose Tool .

B. Persönliche Reports

B.1. Jan Balmer

todo

B.2. Theo Winter

todo

C. Zeit Beurteilung

Einleitung todo

C.1. Total verbrauchte Zeit

todo

C.2. Verbrauchte Zeit pro Monat

todo

C.3. Vebrauchte Zeit pro Aktivität

todo

D. goProbe

GoProbe erlaubt leichtgewichtiges aggregieren von Paketen und deren effiziente Speicherung. Eine Abfrage der gespeicherten Paketen ist via Querying Flows möglich. Da die Installation von goProbe mit dem zur Verfügung gestellten MAKE-file nicht ganz fehlerfrei abläuft gibt es diese Installations-Anleitung.

D.1. Systemvoraussetzungen

Diese Anleitung bezieht sich auf Ubuntu 14.04 x64. Für andere System sollte man sich auf die offizielle Installationsanleitung im GitHub-Repository von goProbe verlassen.

D.2. Installation

Zuerst müssen die fehlenden Dependencies installiert werden. Unter anderem "build-essential, flex, curl und bison".

```
1  $ sudo apt-get install build-essential bison flex curl
```

Danach clonen wir das GitHub-Repository.

```
1  $ git clone https://github.com/open-ch/goProbe.git
```

Nun wechseln wir in das goProbe Verzeichnis, erlangen root-Rechte und führen das MAKE-File mit dem Parameter älläus.

```
$ cd ../git/goProbe
$ sudo su
$ make all
```

Das Make File lädt automatisch die noch fehlenden Dependencies herunter und versucht goProbe zu builden. Wenn der Kompilier-Vorgang fertig läuft dann wurde goProbe erfolgreich installiert. Bei uns hat dass jedoch nicht geklappt, wir mussten hier ins libpcap-Verzeichnis wechseln und dieses zuerst einzeln builden und installieren.

```
$ cd libpcap-1.5.3
$ make clean
$ make install
```

Nun können wir zurück ins Hauptverzeichnis wechseln und goProber erneut builden.

```
$ cd ..
$ make all
```

Der Build-Vorgang zeigt nun zwar einige Build-Warnings sollte aber vollständig durchlaufen. Als letzten Schritt müssen wir jetzt nur noch das Bibliotheken-Verzeichnis "/usr/local/goProbe/lib"korrekt verlinken.

```
$ cd ~
$ vim .bashrc

#Am Ende des Files einfuegen:
export LD_LIBRARY_PATH=/usr/local/goProbe/lib
```

Nun kann ein neues Terminal geöffnet werden und der Befehl 'goProbe' sollte ohne Errors ausgeführt werden können.

E. Jenkins Buildserver

Um repetitive Aufgaben möglichst zu vermindern haben wir Jenkins als CI Server eingesetzt. Wir haben Jenkins so konfiguriert dass für jeden Commit, sowohl beim Programm als auch bei der LaTeX-Dokumentation, ein neuer Build-Job gestartet wird. In diesem Kapitel werden die konkreten Konfigurationen dokumentiert.

E.1. Systemvoraussetzungen

Wir haben Jenkins auf dem HSR VPS mit einem Ubuntu 14.04 x64 betrieben. Dabei wurden folgende Abhängigkeiten vorinstalliert:

- Jenkins 1.610
- libpcap0.8-dev
- go 1.4.2
- git 2.3.5
- texlive-full 2015
- golang lstlang0.sty

TODO: https://bitbucket.org/korfuri/golang-latex-listings/src/6334c5cbf471/lstlang0.sty korrekt attributieren.

E.2. GitHub - WebHook Konfiguration

Um bei jedem Commit direkt einen Build Vorgang auszulösen haben wir bei GitHub für jedes Projekt einen WebHook definiert. Beim Documentation Repository wurden die folgenden Einstellungen gewählt:

Option	Einstellung
Payload URL	http://username:password@152.96.56.53:40000/job/IPSecDiagTool%20-%20Documentation/build
Content type	application/x-www-form-urlencoded
Secret	
Trigger	Just the push event.
Active	true

Die Einstellungen beim IPSecDiagTool Repository sind analog.

E.3. IPSecDiagTool - Go Build Job

E.3.1. Job Einstellungen

Die folgenden Einstellungen wurden beim Go Build Job verwendet:

Option	Einstellung
Discard Old Builds	Yes
Log Rotation	Max # of builds to keep 5
SCM	None
Build Trigger	None
Archive the artifcats	src/github.com/ipsecdiagtool/ipsecdiagtool/ipsecdiagtool

E.3.2. Bash Script

```
1 #!/bin/sh
2 echo "Cleaning workspace"
3 if [ -d src ]; then
4    rm -r *
```

```
5  fi
6
7  export GOROOT=/usr/local/go
8  export GOPATH="$WORKSPACE"
9  export PATH="$PATH:$GOROOT/bin:$GOPATH/bin"
10
11 echo "Downloading dependencies"
12 #sudo apt-get install libpcap-0.8-dev
13 go get code.google.com/p/gopacket
14 go get code.google.com/p/gopacket/pcap
15 go get golang.org/x/net/ipv4
16
17 echo "Moving to program directory"
18 cd src
19 mkdir -p github.com/ipsecdiagtool/
20 cd github.com/ipsecdiagtool
21 git clone https://github.com/IPSecDiagTool/IPSecDiagTool.git
22 mv IPSecDiagTool ipsecdiagtool
23 cd ipsecdiagtool
24
25 echo "Building program"
26 go version
27 go build
```

E.4. Dokumentation - LaTex Build Job

E.4.1. Job Einstellungen

Die folgenden Einstellungen wurden beim LaTeX Build Job verwendet:

E.4.2. Bash Script

Option	Einstellung
Discard Old Builds	Yes
Log Rotation	Max # of builds to keep 5
SCM	Git URL: https://github.com/IPSecDiagTool/Documentation.git Branch: */master
Build Trigger	None
Archive the artifcats	IPSecDiagTool.pdf

```
1 #!/bin/sh
2 pdflatex index.tex > /dev/null 2>&1
3 bibtex index.aux > /dev/null 2>&1
4 pdflatex index.tex > /dev/null 2>&1
5 pdflatex index.tex
6
7 mv -f index.pdf IPSecDiagTool.pdf
```

F. Testumgebung