בס"ד

מגיש: **מיכאל גור**
ת"ז: **207555178**
בית ספר: **הימלפרב, ירושלים**
מנחים: **נתי יחזקאל, סטיב גרוס**

הגשה: **יוני 2017**

תוכן עניינים:

רקע:

כשהיינו צריכים לבחור נושא לפרויקט, בחרתי לעשות משהו כמו Wireshark, כי התעניינתי באיך התוכנה פועלת, ואיך בכלל פרוטוקלים ברשת פועלים וקשורים זה לזה, ואכן, Cablefish פועלת באופן דומה לWireshark, גם אם חסרים הרבה דברים, אך הבסיס קיים ופועל.

התעבורה ברשת פועלת בפקטות שהצדדים שולחים זה לזה. הפקטות בנויות משכבות של פרוטוקולים, כשלכל אחד יש תפקיד אחר והוא פועל ברמה שונה. כשיש בעיות ברשת, צריך לבדוק את הפקטות ולבודד את הבעיה. ההסנפה מאפשרת מעקב אחר הפקטות שעוברות בכרטיס הרשת, ניתוחן ובידוד הבעיה. תוכנות כמו Wireshark מציגות את הפקטות שהוסנפו ומספקות מידע עליהן.

תיאור המוצר:

Cablefish מאפשר הסנפת וניתוח תעבורת רשת. התוכנה מסניפה פקטות רשת, מעבדת אותן ומציגה אותן למשתמש. הפקטות מחולקות לפרוטוקולים השונים ולשדות שהמשתמש יכול לנתח. ניתן לסנן את הפקטות המוצגות על פי שדות או פרוטוקולים מסוימים. בנוסף, התוכנה מזהה שיחות ברשת בין המשתמש ליעדים שונים, ומאפשרת סינון של המידע על פיהן, כך שמתאפשר ניתוח של שיחה מסוימת.

ניתוח המידע מאפשר למשתמש לבדוק את תקינות חיבור הרשת שלו, מאפשר לחברות לבדוק את תקינות השרתים שלהן ולמתכנתים לבדוק את אופן פעילות התוכנה שלהם או הפרוטוקולים שלהם.

את המידע שהתוכנה הסניפה ניתן לשמור בקבצי cf., שמכילים מידע מעובד ומאפשרים פתיחה\ שמירה מהירה על ידי Cablefish, או בקבצי pcap. שנתמכים על ידי תוכנות שונות.

בחרתי ליצור את Cablefish בעיקר לצורכי למידה והבנת אופן הפעילות של תוכנות כמו Wireshark, והיא מבוססת עליהן.

מבט אישי:

כשבחרתי לפתח את Cablefish, ההתעניינות המרכזית שלי לא היתה ביצירת ומימוש רעיון חדש, אלא בתהליך הפיתוח עצמו. התעניינתי בתהליך של יצירת התוכנה מאפס, עד שהיא הפכה למשהו. אולי לא המשהו הכי מדהים או מקורי, אבל משהו פועל.

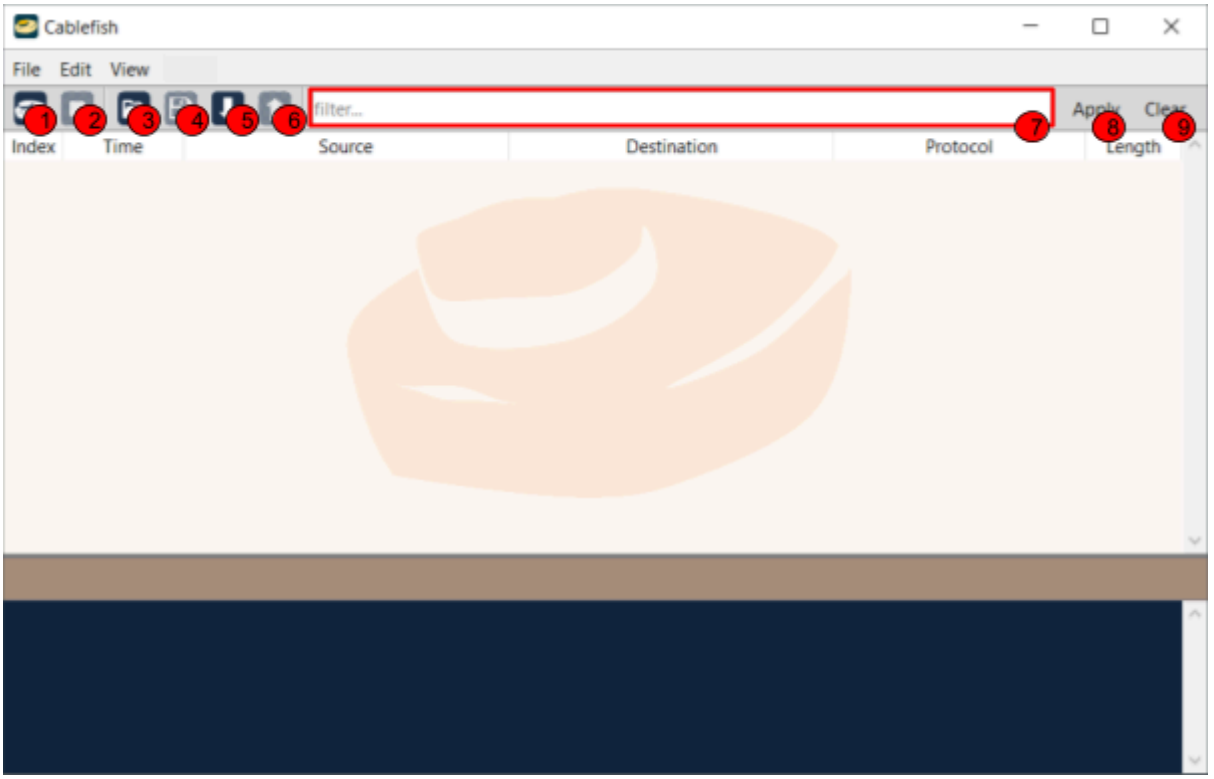
מכיוון שהתמקדתי בעיקר בפיתוח ולא ברעיון, ניסיתי לכתוב תוכנה יציבה עם בסיס נתונים טוב, והאתגר המרכזי היה יצירת המערכת לעיבוד פקטות, שתהיה בעלת ממשק פשוט, אך שגם תהיה גמישה.

הספקתי לממש את רוב מה שרציתי, אך ישנם עוד כמה דברים שתכננתי להוסיף ולא הספקתי, כיוון שהעדפתי להתמקד בקוד הקיים ולשפר אותו:

- בחירת כרטיס רשת. מדובר בפעולה יחסית מסובכת בWindows, והספריות השונות לא פעלו היטב ביחד בנושא הזה.
- Defragmentation של פקטות IP. לא מאוד מסובך אבל לא הספקתי.
- הוספת עוד פרוטוקולים. שוב, לא פעולה מסובכת בגלל האופן שבו מימשתי הגדרת פרוטוקולים.

תובנת בונוס: לא לעשות יותר פרויקטים גדולים בPython וtk. תמיד בלאגן גדול.

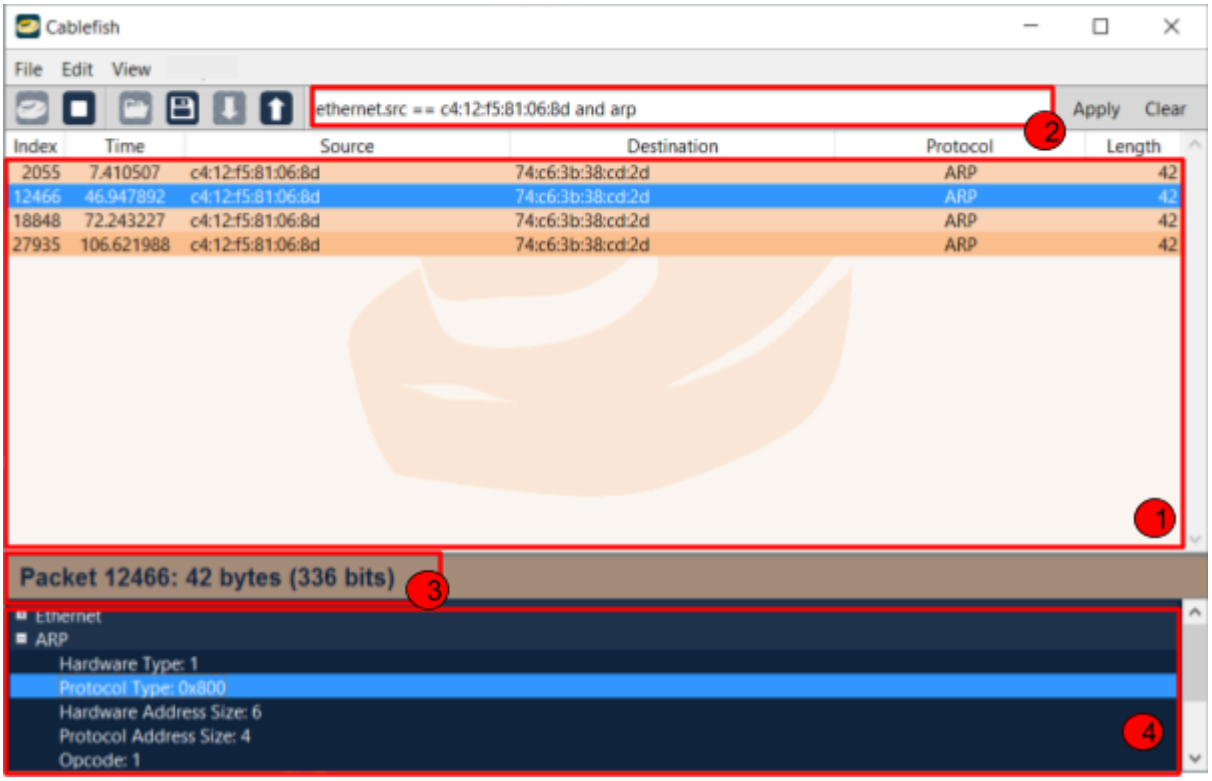
תיאור הממשק למשתמש:



זהו הממשק של Cablefish:

1) התחלת הסנפה חדשה
2) עצירת הסנפה
3) פתיחת קובץ cf.
4) שמירת קובץ cf.
5) יבוא קובץ pcap.
6) יצוא קובץ pcap.
7) עריכת המסנן (filter) להצגת המידע
8) החלת המסנן
9) ניקוי המסנן

כאשר המשתמש מתחיל הסנפה או פותח קובץ, מידע יתחיל להופיע במרכז המסך.



עכשיו המשתמש יכול לעבור על המידע ולנתח אותו.

1) רשימה של הפקטות שהוסנפו. עבור כל פקטה מוצג מידע כמו כתובת המקור, כתובת היעד והפרוטוקול שמשתמש בה.
2) המסנן לפקטות שמוצגות. פקטות שאינן מתאימות למסנן לא יופיעו ברשימה.
3) מידע על הפקטה הנוכחית שהמשתמש בחר.
4) הפרוטוקולים והשדות של הפקטות הנוכחית. ניתן להעתיק את הערך של השדות (Ctrl + c או Edit-> Copy Field Value).

בשלב הזה, אם המשתמש ינסה לצאת מהתוכנה, היא תזכיר לו לשמור את המידע שלו. ניתן לדלג על השלב הזה (Cancel).

<u>המסנן</u>:
ניתן לסנן את המידע שהתוכנה מציגה. זהו הפורמט על פיו המסנן פועל:

כדי לסנן פקטות על פי פרוטוקול מסוים, משתמשים בשם הפרוטוקול. למשל:

התוכנה תציג רק פקטות שמופיע בהן פרוטוקול Ethernet.

כדי לסנן פקטות על פי ערך של שדה מסוים, משתמשים בשם הפרוטוקול, נקודה, ואז שם השדה:

התוכנה תציג רק פקטות שמופיע בהן פרוטוקול TCP ושערך השדה src של TCP הוא 433.

עבור פרוטוקולים שיש להם שדות מקור ויעד (כמו Ethernet, IP…), יש שדה מיוחד בשם conversation_id שמאפשר סינון פקטות על בסיס שיחה.

התוכנה תומכת באופרטורים הבאים:
not, and, or, ==, !=, <, >, <=, >=, ()
כמו כן, היא מזהה כתובות Ethernet, שמיוצגות על ידי מספרים הקסדצימלים שמופרדים על ידי נקודותיים(:), למשל:
AA:11:22:33:44:55
וכתובות IP, שמיוצגות על ידי מספרים שמופרדים בנקודה(.):
172.16.254.1

סביבת העבודה:

Cablefish נכתב ב2.7 Python והוא רץ על Windows. התוכנה משתמשת בWinPcap, שמממש את הספריה Pcap בWindows. הספריה Pcap מאפשרת הסנפת פקטות, והשתמשתי בה כיוון שאי אפשר לעשות זאת בwindows בלי הרשאות מנהל. התוכנה משתמשת בPyPcap, שמאפשרת שימוש של WinPcap בשפת Python. הממשק של Cablefish נכתב בספרייה Tkinter.

פיתחתי את Cablefish בסביבת הפיתוח Intellij של Jetbrains, שנכתבה במקור לשפת Java אבל יש לה תוסף לתמיכה בPython.

בדיקת התקינות של הקוד נעשתה באופן ידני. הפונקציות, המאפיינים של הממשק והקבצים שהתוכנה פותחת או מייצרת נבדקו.

תיאור המודולים:

<u>מודולים מובנים:</u>

- Tkinter- ממשק בPython שמאפשר עבודה עם tk, ספריית GUI פשוטה.
- threading- ממשק לניהול threads (תהליכונים) בPython.
- pickle- מודול שמממש סריאליזציה ודסריאליזציה של אובייקטים בPython.
- datetime- ספרייה בפייתון שמאפשרת עבודה עם זמנים, תאריכים וtimestamps.
- binascii- מודול שמאפשר המרה של טקסט למידע בינארי.
- re- מודול שמאפשר עבודה עם ביטויים רגולריים בPython.
- ast- מודול שמשמש לעיבוד קוד Python ויכול לבדוק תקינות תחבירית של קטעי קוד.
- Queue- מודול שמממש אוביקט Queue בPython.
- math- מודול שמכיל פונקציות מתמטיות שונות.
- collections- מודול שמכיל Containers מסוגים שונים.
- importlib- ספרייה שמאפשרת יבוא דינאמי של מודולים בPython.
- os- ספרייה שמאפשרת גישה לפונקציות שונות של מערכת ההפעלה.
- json- ספרייה שמאפשרת עבודה עם קבצי JSON.

<u>מודולים פומביים:</u>

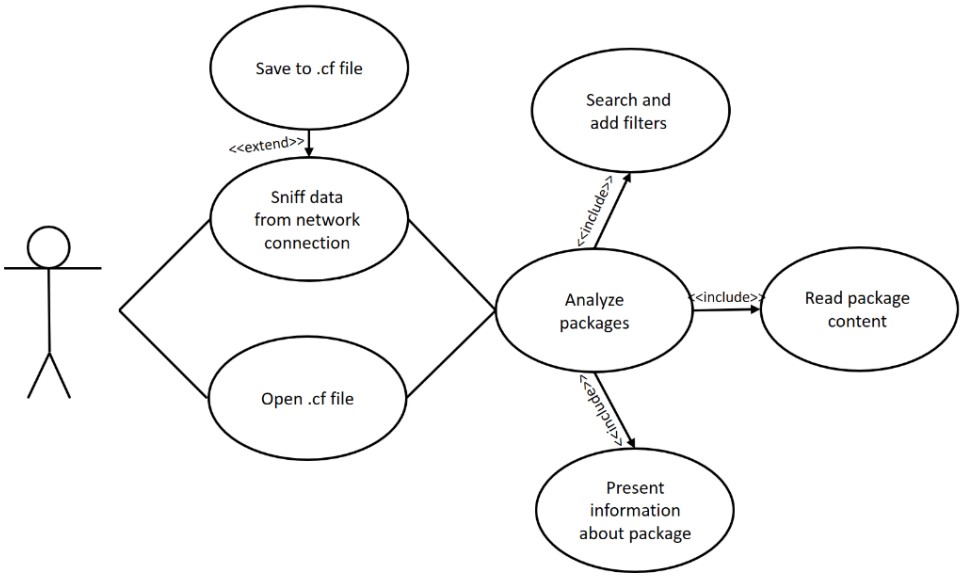
- pypcap- ממשק בPython לספריה WinPcap, שמאפשרת הסנפת תעבורת רשת במערכת ההפעלה Windows.

<u>מודולים שפותחו:</u>

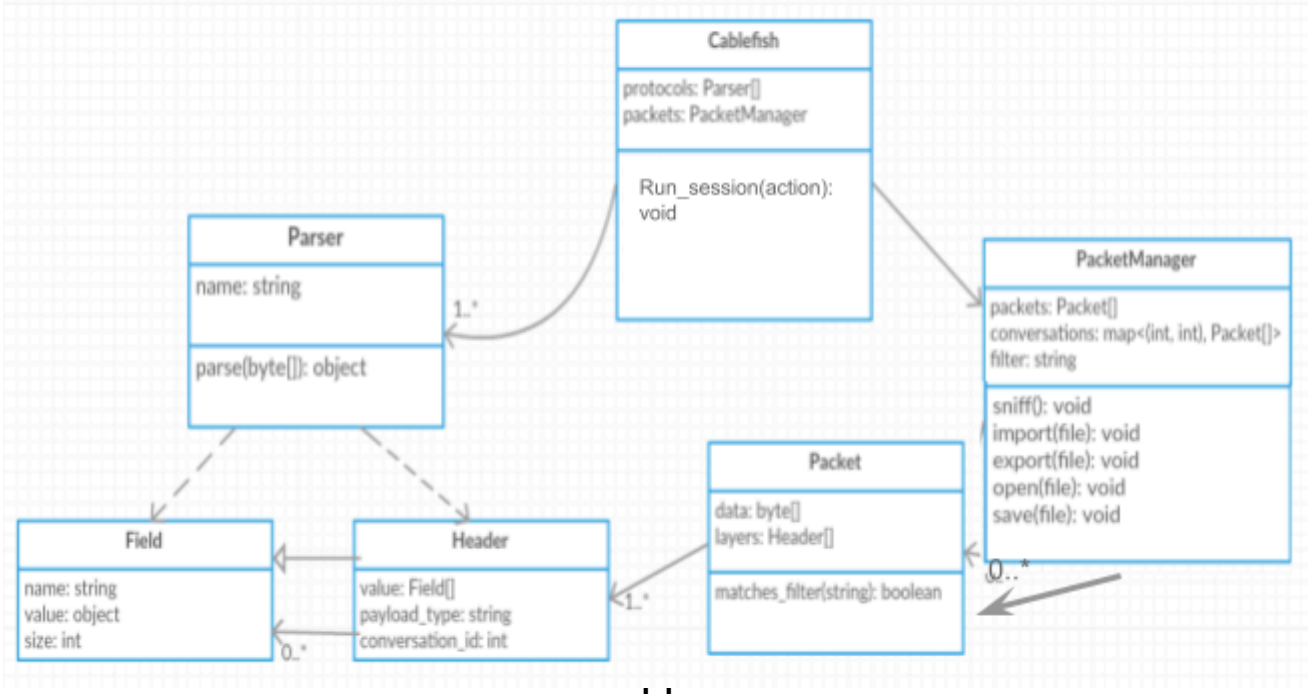
- protocols- מודול שמייבא מידע על עיבוד פקטות שנמצא בקבצי Json בתיקיה protocols, ומייצר אובייקטים מסוג Parser שהתוכנה יכולה להשתמש בהם.
- filter- מודול שמקבל מסננים מהמשתמש והופך אותם לקוד שפקטות יכולות להריץ ולבדוק אם הן מתאימות למסנן.
- export_pcap- מודול שמאפשר יצוא של קבצי libpcap (מסוג pcap.).
- gui– חבילה הכוללת את כל מודולי הממשק הגרפי, שממומשים בTkinter.

תיעוד הקוד:

Use-Case Diagram של הפרויקט:



Class-Diagram של הפרויקט:



מבנה נתונים (נלקח חלקית ממסמך העיצוב):

Cablefish היא המחלקה הראשית שמייצגת את התוכנית. המחלקה אחראית לקביעת מבנה הפקטות, לניהולן, וניהול הפעולה של התוכנה. בנוסף, Cablefish מנהלת את ה-ui. בתחילת ריצתה של התוכנית, Cablefish מייבאת קבצי JSON שמתארים את המבנה של פרוטוקולים שונים ומגדירים את המבנה של הפקטות, והיא יוצרת מהם Parserים.

בעקבות פעילות המשתמש, הui שולח לCablefish פקודות שונות כמו התחלת הסנפה, פתיחת ושמירת קבצים, וכו'. Cablefish תריץ את הפקודות על thread נפרד. את קבלת וניהול המידע מבצע הPacketManager, עליו אפרט בהמשך.

המידע הנא של הפקטות עובר לParserים שתפקידם ליצור אובייקט Packet עם מידע מסודר ומחולק לשדות שהתוכנה והמשתמש יכולים להבין. Cablefish יוצרת Parser עבור כל פרוטוקול, שמכיל Parserים עבור כל שדה בכותרת שלו. כל Parser כזה יודע לעבד מידע וליצור Field (שדה) או Header (כותרת) עם מידע שמיש. כל Field מכיל מידע מסוים, למשל השם שלו, הערך שהוא קיבל מהParser והגודל שלו בבתים. גם Headerים מכילים את המידע הזה, כשהערך שלהם הוא רשימה של השדות שמרכיבים אותם. בנוסף, כל Header צריך לדעת מה סוג הפרוטוקול הבא (אם יש כזה), ואם מדובר בפרוטוקול שיחות שלא מתבסס על הפרוטוקולים לפניו, ב-Header יש גם מצביע לשיחה כולה שבה הפקטה נמצאת. כשלא נמצא הפרוטוקול הבא, התוכנה מניחה שהמידע שנותר הוא ההודעה שהועברה בפקטה, ונוצרת Packet, שמכילה את כל הHeaderים שמרכיבים אותה עם המידע המעובד שבהם, ואת כל המידע הנא, בבתים. הPacket נשמרת בPacketManager של Cablefish.

הPacketManager שומר בתוכו את כל הפקטות שעובדו. כשנוספת אליו Packet, הוא עושה לה עיבוד אחרון, והוא מזהה שיחות בין פקטות. התפקיד הנוסף של הPacketManager הוא לתקשר עם הgui של Cablefish, ולהציג את הפקטות שבו. כיוון שעיבוד הפקטות והgui פועלים בthreadים שונים, PacketManager

משתמש בQueue מיוחד, שהוא thread safe. המשתמש יכול לקבוע filter (מסנן) לפקטות, שהPacketManager משתמש בו ומציג רק פקטות תואמות.
לבסוף, המשתמש יכול לשמור את הפקטות בפורמט cf. או לייצא אותן לpcap.,
וCablefish תשתמש במידע שבPacketManager ותבצע זאת. Cablefish לא כוללת בסיס נתונים, ועל אחריות המשתמש לשמור את המידע שהוצג בתוכנה.

האלגוריתמים המרכזיים בפרויקט:

<u>עיבוד המידע:</u>

פקטות רשת מחולקות לפרוטוקולים, שכל אחד מהם בנוי משדות שונים, ומכיל את הפקרוטוקול הבא. Cablefish מקבלת פקטות לא מעובדות, כbuffer של בתים, והיא צריכה לחלק אותן לשדות שיאפשרו ניתוח וסינון שלהן. כל פרוקטול רשת בנוי באופן שונה, וכל שדה מכיל מידע שונה, לכן דרכי העיבוד משתנות משדה לשדה ומפרוטוקול לפרוטוקול. עם זאת, ישנם גורמים משותפים שנחוצים לכל השדות, כמו:

- שם (id שהתוכנה יכולה להשתמש בו).
- גודל בבתים.
- פונקציה שמסוגלת לקחת מידע ולעבד אותו לערך.
- ערך, לאחר העיבוד
- דרך להצגת ערך השדה (המרת הערך לstring).

המידע הזה נחוץ גם לכותרות של פרוטוקולים, אך לכותרות יש ערכים נוספים שמשותפים לרובן, כמו:

- תיאור רשימת השדות שמרכיבים אותן.
- פרוטוקול "אב", שיכול להכיל את הפרוטוקול הנוכחי כמטען (payload).
- מידע על המטען של הפרוטוקול הנוכחי.
- בחלק מהפרוטוקולים, מידע על נקודות הקצה (endpoints), על מנת שיתאפשר זיהוי שיחות.

אם כן, נחוץ מבנה נתונים ש:

- יודע לקבל מידע נא ולהפוך אותו לפקטה מעובדת.
- מנצל את העובדה שהמבנים של השדות השונים דומים.
- יודע לדאוג לכל שדה בנפרד, כי המבנה שלהם לא זהה. העיבוד של חלק מהשדות שונה מאוד מזה של אחרים, למרות הבסיס המשותף.
- יודע להסיק מהמידע שמופיע בפרוטוקול מה סוג המטען שלו.
- ואם אפשר, שיהיה גמיש, ויאפשר מימוש זריז של עיבוד של פרוטוקולים רבים ושונים.

כדי לענות על הצרכים האלה, בחרתי במבנה נתונים שמבוסס על קלאס בשם Parser. ישנו Parser לכל שדה ולכל כותרת, כיוון שהמבנה שלהם דומה, אך ניתן להוסיף לכל אחד מידע ופונקציות נוספים באופן גמיש, כך שאותו אובייקט יכול לטפל בהרבה מקרים שונים. כל Parser שמקבל מידע יוצר מעין מילון, שייצג את השדה ומכיל את כל המידע הנחוץ כדי לנתח ולהציג אותו. Parser שמייצג כותרת, מכיל Parserים שמייצגים שדות, והערכים שהם מייצרים מוכנסים לערך שהParser הראשי מייצר. כל Parser שמייצג כותרת אחראי לספק את השם של הפרוקוטול שנמצא במטען, אם הוא זיהה כזה.

נוצר מצב שעבור כל פרוטוקול ועבור כל שדה היה צריך לכתוב אובייקט מיוחד שמרחיב את Parser. המבנה ענה על רוב הדרישות, אך הוא נראה מגושם, והיו בו חזרות רבות על אותו הקוד. לכן החלטתי שיותר מתאים להגדיר את האובייקטים האלה בקבצי JSON, שמגדירים את כל הערכים הנחוצים של הParserים, והם אינם חלק מהקוד עצמו. אלה היתרונות של הפורמט הזה:

- גושי קוד מגושמים וחוזרים על עצמם לא נמצאים יותר בקוד של הפרויקט, אלא כקבצים נפרדים.
- יותר פשוט ליצור דרכים לפישוט הגדרות הParserים, ופיחות החזרות על הקוד.
- שדות רבים מתייחסים לאותן פונקציות ואותם ערכים, כך שנחסך מקום.
- העריכה של הקבצים האלה פשוטה בהרבה, וגם גמישה בהרבה. יותר פשוט להגדיר עצמים בJSON מאשר לכתוב קוד.
- המבנה של הפרוטוקולים יותר ברור לעין, ולא מסתיר אותו קוד.

עם זאת, יש לכך גם מספר חסרונות:

- אי אפשר לכלול קוד בקבצי JSON, לכן כל הפונקציות נמצאות בקבצי python נפרדים. הקבצים צריכים להזכיר אותן בשמן. אמנם הדבר הופך את המבנה של הפרוטוקולים ליותר ברור, אך הוא קצת מגושם, וכשCablefish מייבאת את הקבצים היא צריכה לחבר אליהם את הפונקציות.
- תקשורת בין קבצי JSON שונים מסובכת יותר, בהשוואה לקוד בו פשוט אפשר לציין אובייקטים שונים.

בכל פעם שהתוכנה רצה, היא פותחת את קבצי הJSON ושומרת את הParserים שהם מגדירים. כשמתקבלת פקטה חדשה, היא עוברת לטיפולם. אחרי שכל Parser של פרוטוקול מסיים את עבודתו, המידע נשמר באובייקט הPacket, והParser נותן את השם של הParser של הפרוטוקול הבא. אם לא נמצא פרוטוקול, התוכנה מניחה שהמידע שנשאר הוא ההודעה, ואובייקט הPacket מוכן. האובייקט מועבר

לPacketManager, שבודק את הendpoints בפרוטוקולים השונים ומשבץ את ה-Packet בשיחות שונות.

<u>סינון הפקטות:</u>

בעיה נוספת ביצירת התוכנה היתה סינון הפקטות. צריך לאפשר למשתמש להכניס מסננים פשוטים, ועל התוכנה להבין אותם וליישם אותם. הדרישות מהמסננים היו כאלה:

- צריך לזהות שמות פרוטוקולים, ולהציג רק פקטות שמכילות את הפרוטוקולים
- צריך לזהות גישה לשדות של פרוטוקולים, ולהחזיר את ערכם
- צריך לאפשר שימוש באופרטורים בסיסיים, כמו: ==, and…
- צריך לזהות ערכים שונים, ביניהם כתובות MAC ו- IP.

כדי להימנע מכתיבת עיבוד אמיתי של המסננים, כיוון שמדובר בנושא מסובך שיכול להוות פרויקט בפני עצמו, בחרתי להשתמש בeval של Python. הפונקציה eval מקבלת string, מתייחסת אליו כקוד Python ומבצעת אותו. הפונקציה הזאת מסוכנת, שכן המשתמש יכול להריץ דרכה מה שירצה, אך העיבוד של המסנן שהמשתמש מכניס לא מאפשר הרצת כל קוד. הפונקציה eval כפי שהיא מאפשרת שימוש באופרטורים, אך המסנן של המשתמש עוד אינו מוכן להרצה כקוד.

לקלאס Packet יש פונקציה בשם hasattr, שמקבלת שם של פרוטוקול, או שם של שדה בפורמט הבא:

<protocol>.<field>

הפונקציה מחזירה את ערך השדה\ הפרוטוקול אם הוא נמצא, וNone אם לא נמצא. גם המסנן שהמשתמש מכניס נכתב בפורמט הזה. כל מה שנשאר לעשות הוא להחליף משתנים כמו:

Ethernet.ethertype

בפקודה שניתנת לביצוע ע"י interpreter של Python, כמו:

self.hasattr('Ethernet.ethertype')

המודול filter אחראי לבצע את הדבר הזה. באמצעות regular expressions, הוא מזהה במסנן של המשתמש מזהים (Identifiers) ומחליף אותם בפקודה כזאת. המודול גם מזהה ערכים כמו כתובות MAC ו- IP, וממיר אותם מהייצוג הטקסטואלי שלהם לערך שהתוכנה יכולה לחשב. לאחר שfilter עיבד את המסנן, המודול ast אחראי לבדוק שהקוד שנוצר תקין. במקרה של קוד בלתי תקין, התוכנה מתעלמת מהמסנן, וכשהקוד תקין, המסנן החדש מוכנס לPacketManager, וכל Packet משתמש בו ובeval כדי לראות אם היא מתאימה.

תדפיס הקוד:

Cablefish.py:

```
from threading import Thread
from cf import protocols
from cf.packet_manager import PacketManager
from gui.gui import CFGUI


class CableFish(object):
    """The CableFish application."""
    def __init__(self):
        """
        Create and run a new application.
        """
        self.protocols = protocols.import_from('./protocols/')
        self.packets = PacketManager(self)
        self.sniffer = None
        self.session_running = False

        self.root = CFGUI(self)
        self.root.mainloop()

    def new_session(self, action):
        """
        Utility method, used to start a new capture session on a Thread.

        :type action: function
        :param action: The function to execute on the Thread.
        """
        if self.sniffer and self.sniffer.isAlive():
            self.session_running = False
            self.packets.running = False
            while self.sniffer.isAlive(): pass
        self.sniffer = Thread(target=action)
        self.sniffer.start()
        self.session_running = True
        self.root.capture_menu.update()
        self.root.after(1, lambda: self.root.update_packets(self.packets.queue))

    def update_filter(self, filter):
        """
        Change the display filter.

        :type filter: str
        :param filter: The new display filter
        """
        self.packets.set_filter(filter)
        self.root.clear_packets()
```

```
    def quit(self):
        """
        Ask user to save the capture, then close everything and quit safely.
        """
        if self.packets.saved or not self.packets.packets:
            self.root.quit()
            self.packets.stop()
            return
        self.root.asksave()
        self.packets.stop()

# run the application
if __name__ == '__main__': app = CableFish()
```

Protocols.py:

```
import json
import os
from collections import OrderedDict
from importlib import import_module
from cf import fields
from cf.fields import Parser


# dict[str, Parser]: Stores the imported parsers. Access using protocol ID. """
protocols = {}

# dict[str, dict]: Types are defined in the JSON files, and are used as
base-classes for parsers.
types = {}

# dict[str, module]: Python modules that have functions that the parsers use.
modules = {}

# set[str]: Used to keep track of which files we've imported.
imported = set()


def import_from(dir):
    """
    Imports all files from specified directory.

    :type dir: str
    :param dir: Name of directory of files to import.
    :rtype: dict[str, Parser]
    :return: Dictionary contains imported parsers.
    """
    for i in os.listdir(dir):
        if i.split('.')[-1].lower() == 'json':
            import_file(dir, i)

    return protocols


def import_file(dir, name):
    """
    Imports a JSON file from a directory, and generates Parser objects (which are
then inserted into protocols).

    :type dir: str
    :param dir: Parser file location.
    :type name: str
    :param name: Name of parser JSON file.
    """
    if name in imported: return
    imported.add(name)

    file = json.loads(open(dir + name, 'r').read())
```

```
    # If file depends on other files' data, import them first
    for fn in file.get('using', []): import_file(dir, fn)  # possible TODO: remove
file dependency

    # Import python code that the Parser will use
    for m in file.get('include_modules', []):
        if m in modules: continue
        modules[m] = import_module('protocols.python.' + m)

    # Import types
    for t in file.get('types', []):
        if 'base' in t:
            t = dict(types[t['base']], **t)
            t.pop('base')
        types[t['name']] = t

    # Generate Parser and put it in protocols
    if 'protocol' in file:
        file['protocol']['children'] = []  # fixme: does not belong here
        protocols[file['protocol']['id']] =
Parser(**fix_JSON_data(file['protocol']))


# dict. Default values for required attributes of parsers.
default = dict(size=0, name='', id='', bit_value=False, str=str, parse=lambda _,
__: None)


def fix_JSON_data(data):
    """
    Fixes raw JSON data so it can be used by the Parser.

    :type data: dict[str, object]
    :param data: JSON data to be fixed.
    :rtype: dict[str, object]
    :return: Fixed data.
    """

    # Use values from type
    if 'type' in data:
        data = dict(types[data['type']], **data)
        data.pop('type')

    # Add missing attributes that are required
    data = dict(default, **data)

    # If field is a header, parse its fields
    if 'header' in data:
        data['parse'] = fields.parse_header
        fields_dict = OrderedDict()
        for f in data['header']:
            fields_dict[f['id']] = Parser(**fix_JSON_data(f))
        data['header'] = fields_dict
```

```
    # Parse payload data
    if 'payload' in data:
        data['payload'] = Parser(**fix_JSON_data(data['payload']))

    # Replace function names with actual functions from modules.
    for k in data.keys():
        if k.startswith('f_') and not callable(data[k]):
            data[k[2:]] = get_func(data.pop(k))

    return data


def get_func(name):
    """
    Get a function from the one of the imported modules

    :type name: str
    :param name: Name of the function. Format: '<module>.<function>'
    :rtype: function
    :return: The imported function
    :exception: If the module does not exist or does not have the function.
    """
    s = name.split('.', 1)
    return getattr(modules[s[0]], s[1])
```

Fields.py:

```
from collections import OrderedDict
from math import ceil

import protocols


class Parser(object):
    """
    Used to parse a specific Header or Field from raw packet data.
    """
    def __init__(self, **kwargs):
        """
        Create a new parser.

        :param kwargs: Parsing options
        """
        self.args = kwargs
        functions = {}
        for k in kwargs.keys():
            if k.endswith('_func'):
                functions[k[:-5]] = self.args[k]
                del self.args[k]
        self.args['functions'] = functions

        parents = []
        if 'parent' in self.args:
            parents.append(protocols.protocols[self['parent']])
        elif 'parents' in self.args:
            for p in self.args['parents']:
                parents.append(p)
        for p in parents:
            if not hasattr(p, 'children'):
                setattr(p, 'children', [])
            p.children.append(self)

    def __getitem__(self, item):
        return self.args[item]

    def __setitem__(self, key, value):
        print key, value
        self.args[key] = value

    def parse_main(self, data, offset, packet):  # TODO: rename
        """
        Parse raw packet data using Parser's arguments.

        :type data: buffer
        :param data: The raw packet data.
        :type offset: int
        :param offset: The offset of the relevant data in the packet.
        :type packet: Packet
        :param packet: The Packet object.
```

```
        :rtype: Field|Header
        :return: The parsed data, in a Field object that contains additional info.
        """
        res_dict = self.args.copy()

        if 'functions' in self.args:
            res_dict.pop('functions')
            for f in self.args['functions']:
                res_dict[f] = self.args['functions'][f](packet)

        res_dict['offset'], res_dict['parent'] = offset, packet

        if 'header' in self.args:
            res_dict['parser'] = self
            res = Header(res_dict, data)
            delattr(res, 'header')

            if 'endpoints' in self.args:
                res.endpoints = (getattr(res, 'src'), getattr(res, 'dst'))

        else: res = Field(res_dict, data)

        return res


class Field(object):
    """
    Represents a field of a protocol in the Packet.
    """
    def __init__(self, args, data):
        """
        Create a new Field, using raw data and info from the Parser.

        :type args: dict
        :param args: Information about the Field.
        :param data: The raw packet data.
        """
        self.__dict__ = args
        self.value = self.parse(self, data)
        delattr(self, 'parse')

    def __repr__(self):
        """
        Represent the field as str. What this function does changes per Field.

        :return: The textual representation of the Field.
        """
        return self.str(self.value)


class Header(Field):
    """
    Represents the header of a protocol in the Packet.
    """
```

```
    def __getattr__(self, item):
        """
        Enables access to the values of this Header's Fields.

        :type item: name
        :param item: The name of the field.
        :return: The value of the field.
        :raise AttributeError: If item is not an attribute of the header or a Field.
        """
        if 'value' in self.__dict__ and item in self.value:
            return self.value[item].value
        raise AttributeError(repr(item))

    def __repr__(self):
        """
        Represent the header as str. Requires an additional function since an f_str
is not defined for headers.
        Unused.

        :rtype: str
        :return: The textual representation of the Header and its Fields.
        """
        return '{\n\t%s\n}' % \
        (',\n\t'.join('%s: %s' % (f.name, repr(f).replace('\n', '\n\t')) for f in
self.value.values() if repr(f)))

    def get_payload(self):
        """
        Gets the type of this protocol's payload, if found.

        :rtype: str
        :return: The ID of the next protocol in the Packet (if exists).
        """
        if not hasattr(self, 'payload'): return None
        # Convert self.payload from a Parser object to the ID (str) of the payload.
        self.payload.args['parent'] = self
        self.payload = self.payload.args['func'](self.payload)
        return self.payload


def parse_header(header, data):
    """
    Used to parse Headers (or Fields that have Fields of their own (treats them as
Headers)).

    :type header: Field
    :param header: The Field object containing the parsing information.
    :type data: buffer
    :param data: The raw packet data.
    :rtype: OrderedDict[str, Field]
    :return: The list of Fields in the Header (can also be accessed using IDs).
    """
    header.value = OrderedDict()
    # Parse all Fields in the Header. f is the current Field's Parser.
```

```
    for f in header.header.values():
        # Bit offsets are represented by 0.125 per bit.
        # If the field doesn't support bit offsets, move to the next byte.
        if not f['bit_value']: header.size = int(ceil(header.size))
        # Generate a Field object using f.
        v = f.parse_main(data, header.offset + header.size, header)
        # Put the Field object in the header, and increase its size
        header.value[f['id']] = v
        header.size += v.size
    return header.value
```

Packet.py:

```
from collections import OrderedDict
from datetime import datetime


class Packet(object):
    """
    Represents a captured packet. Contains packet data and protocols.
    """
    def __init__(self, index, ts, data, manager):
        """
        Create a new Packet.

        :type index: int
        :param index: Packet's index in the capture.
        :type ts: int
        :param ts: Timestamp.
        :type data: buffer
        :param data: The raw data of the packet.
        :type manager: PacketManager
        :param manager: The Packet Manager. Used for detecting conversations &
general capture info.
        """
        self.layers = []
        self.index = index
        self.raw_data = str(data)
        self.size = 0
        self.ts = ts
        self.time = (datetime.fromtimestamp(ts) -
manager.capture_start).total_seconds()
        p = manager.protocols['ethernet']
        res = None

        while p:
            self.layers.append(p['id'])
            res = p.parse_main(data, self.size, res)
            setattr(self, p['id'], res)
            self.size += res.size

            if hasattr(res, 'endpoints'):
                self.src, self.dst = res.value['src'], res.value['dst']
                e = frozenset(res.endpoints)

                if p['id'] not in manager.conversations:
                    manager.conversations[p['id']] = OrderedDict()
                c = manager.conversations[p['id']]
                if e not in c: c[e] = []
                c[e].append(res)
                res.conversation_id = manager.conversations[p['id']].keys().index(e)
                res.conversation = manager.conversations[p['id']][e]

            p = res.get_payload()
            if isinstance(p, unicode): p = manager.protocols[p]
```

```
        self.protocol = res.name

    def getattr(self, name):
        """
        Used to access protocols and fields.

        :type name: str
        :param name: The name of the required protocol. To access protocol fields,
use: "<protocol-name>.<field-name>"
        :return: The value of the field, or the protocol's Header object.
        """
        t = self
        for i in name.split('.'):
            if not hasattr(t, i): return None
            t = getattr(t, i)
        return t

    def matches(self, filter):
        """
        Check if Packet matches this filter.

        :type filter: str
        :param filter: The filter to check.
        :rtype: bool
        :return: whether the packet matched the filter.
        """
        return eval(filter) if filter else True
```

PacketManager.py:

```
import ast
import pickle
from Queue import Queue
from datetime import datetime
from pcap import pcap
from threading import Lock
from cf.filter import fix_filter
from packet import Packet
import export_pcap


class PacketManager(object):
    """
    Manages captured packets, and keeps track of conversations.
    Used to pass packets to the GUI.
    """

    def __init__(self, app):
        """
        Create a new PacketManager

        :type app: CableFish
        :param app: The application that controls this PacketManager.
        """

        self.app = app
        self.protocols = app.protocols
        self.conversations = {}
        self.capture_start = None

        self.packets = []
        self.queue = Queue()
        self.lock = Lock()
        self.thread = None

        self.filter = ''
        self.filter_valid = True

        self.saved = True
        self.running = False
        # pickle: packets, filter (if valid), capture start

    def add(self, p):
        """
        Add a Packet object to this PacketManager, and queues its display.

        :type p: Packet
        :param p: The Packet
        :rtype: Packet
        :return: The added Packet
        """
        with self.lock:
```

```
            self.packets.append(p)
            if p.matches(self.filter):
                self.queue.put(p)
        return p

    def set_filter(self, filter):
        """
        Set the display filter for the Packets.

        :type filter: str
        :param filter: The user-inputted filter
        """
        # Fix the user's input to be usable
        self.filter = fix_filter(filter)

        try:
            # Check if the filter is valid code. If it's not, ignore it.
            ast.parse(self.filter)
        except:
            self.filter_valid = False
            self.filter = ''
        else:
            self.filter_valid = True

        # Only display Packets that match the filter.
        with self.queue.mutex:
            self.queue.queue.clear()
        for p in self.packets:
            if p.matches(self.filter): self.queue.put(p)

    def sniff(self, name=None):
        """
        Sniff network traffic using PCAP, and organize captured data in Packet
objects.

        :type name: str
        :param name: The id of the network interface, or the name of the .pcap file
(defaults to None).
        """
        self.running = True
        self.saved = False
        self.capture_start = datetime.now()
        for i, (timestamp, packet) in enumerate(pcap(name=name)):
            if not self.running: return
            self.add(Packet(i, timestamp, packet, self))

    def stop(self):
        """Stop the current capture."""
        self.running = False

    def save(self, filename):
        """
        Saves the Packets in a .cf, using Python's pickle format.
```

```
        :type filename: str
        :param filename: Name of the .cf file to save the data in.
        """
        self.running = False
        with self.lock:
            pickle.dump(self.packets, open(filename, 'wb'))
            self.saved = True
        print 'saved packets in', filename

    def open(self, filename):
        """
        Open a .cf file and add its content.

        :type filename: str
        :param filename: The name of the .cf file to open.
        :rtype: List[Packet]
        :return: The Packets from the opened .cf file.
        """
        for p in pickle.load(open(filename, 'rb')): self.add(p)
        return self.packets

    def export_pcap(self, filename):
        """
        Export Packets to a .pcap file.

        :type filename: str
        :param filename: The name of the .pcap file to export data to.
        """
        export_pcap.generatePCAP(self.packets, filename)

    def import_pcap(self, filename):
        """
        Import Packets from a .pcap file.

        :type filename: str
        :param filename: The name of the .pcap file to import data from.
        """
        try: self.sniff(filename)
        except TypeError: pass  # There's a bug in pypcap's code that raises an
Exception for no reason

    def clear(self):
        """
        Delete all Packets in this PacketManager.
        """
        self.packets = []
        self.conversations = {}
        with self.queue.mutex:
            self.queue.queue.clear()
```

Filter.py:

```
import re

pattern_MAC = re.compile('[:-]'.join('[0-9a-fA-F]{1,2}' for _ in range(6)))
pattern_IP = re.compile('\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}')
pattern_IDENTIFIER =
re.compile('(?<!\w)(([a-zA-Z_][a-zA-Z0-9_]*\.)*[a-zA-Z_][a-zA-Z0-9_]*)')


def address_replace(match, split, base):
   return '+'.join('chr(%i)' % int(i, base=base) for i in re.split(split,
match.group()))


def ip_replace(match): return address_replace(match, '\.', 10)


def mac_replace(match): return address_replace(match, '[:-]', 16)


def identifier_replace(match):
   m = match.group()
   if m in ['and', 'or', 'not', 'chr']: return m

   return 'self.getattr("%s")' % m


def fix_filter(filter):
   """
   Converts a filter given by the user into usable code.

   :type filter: str
   :param filter: The filter.
   :rtype: str
   :return: The fixed filter.
   """
   filter = filter.lower()
   filter = re.sub(pattern_MAC, mac_replace, filter)
   filter = re.sub(pattern_IP, ip_replace, filter)
   return re.sub(pattern_IDENTIFIER, identifier_replace, filter)
```

Export_Pcap.py:

```
import binascii
import calendar
from datetime import datetime

pcap_global_header = ('D4 C3 B2 A1'
                      '02 00'  # File format major revision (i.e. pcap <2>.4)
                      '04 00'  # File format minor revision (i.e. pcap 2.<4>)
                      '00 00 00 00'
                      '00 00 00 00'
                      'FF FF 00 00'
                      '01 00 00 00')

pcap_packet_header = ('AA 77 9F 47'
                      '90 A2 04 00'
                      'XX XX XX XX'  # Frame Size (little endian)
                      'YY YY YY YY')  # Frame Size (little endian)


def write_header_bytes(bytestring, f):
    f.write(binascii.a2b_hex(''.join(bytestring.split())))


def write_int(i, f):
    hex_str = '%08x' % i
    f.write((hex_str[6:] + hex_str[4:6] + hex_str[2:4] + hex_str[:2]).decode('hex'))


tz_correction = datetime.fromtimestamp(calendar.timegm((1970, 1, 1, 0, 0, 0)))


def generatePCAP(packets, filename):
    """
    Exports packets into a libpcap file.

    :type packets: list[Packet]
    :param packets: The packets to export.
    :type filename: str
    :param filename: The name of the libpcap file.
    """

    f = open(filename, 'wb')
    write_header_bytes(pcap_global_header, f)
    for p in packets:
        time = datetime.fromtimestamp(p.ts)
        write_int((time - tz_correction).total_seconds(), f)
        write_int(time.microsecond, f)
        write_int(len(p.raw_data), f)
        write_int(len(p.raw_data), f)
        f.write(p.raw_data)
```

מקורות:

הויקיפדיה של wireshark, שסיפקה מידע על האופן בו wireshark מעבד פקטות:
https://wiki.wireshark.org/
אתר effbot, שמכיל מידע על כל מה שקשור לTkinter:
http://effbot.org/tkinterbook/
אתר tcpipguide, שמכיל מידע מפורט על חלק מהפרוטוקולים שמופיעים בCablefish:
http://www.tcpipguide.com/
ויקיפדיה כמובן, שמכילה את המבנים של הכותרות כל הפרוטוקולים שמופיעים בCablefish:
https://en.wikipedia.org/wiki/Internet_protocol_suite/
דף הgithub של pypcap, שמכיל גם דוגמאות לשימוש במודול:
https://github.com/dugsong/pypcap/
הדוקומנטציה הרשמית של Python, שסיפקה מידע על הספריות built-in השונות:
https://docs.python.org/2.7/