LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

ETHERNET TUNELĒŠANAS OPTIMIZĀCIJA, IZMANTOJOT IP PAKEŠU ĪPATNĪBAS

MAĢISTRA DARBS

Autors: **Adrians Heidens**
Stud. apl. Nr. ah08058

Darba vadītājs: prof. Guntis Bārzdiņš

RĪGA 2012

ANOTĀCIJA

Darbā ir izstrādāts un aprakstīts jauns, eksperimentāls tīkla tunelēšanas protokols Zero, kas ir efektīvāks par citiem pieejamiem šādas tunelēšanas risinājumiem. Šis risinājums, atšķirībā no citām pieejamām tīkla tuneļa realizācijām, piemēram, OpenVPN, pie zināmas tīkla konfigurācijas, spēj nodrošināt IP protokola komunikāciju nepalielinot paketes datu apjomu. Tas ļauj izvairīties no IP pakešu sadalīšanas, jeb fragmentēšanas, kā arī no liekas informācijas pārsūtīšanas tīklā. Samazināt pārsūtāmo pakešu skaitu un apjomu ir būtiski tādos tīkla savienojumos, kā satelīta komunikācija, kur sakaru kvalitāte ir ierobežota. Darbā ir aprakstīta protokola ideja, kā arī demonstrēta pilna tā realizācija.

Atslēgas vārdi: datortīkls, ethernet, IP, TUN/TAP, *bridge*, *socket*, programmēšana, Unix, Linux, Python, OpenVPN.

ANNOTATION

Ethernet tunneling optimization using IP packet characteristics

This paper describes the idea and implementation of a new experimental network tunneling protocol Zero which is more effective than other available options. This solution unlike other for example OpenVPN can provide zero-overhead IP protocol tunneling for a relatively small cost. This prevents IP packet fragmentation as well as provides the same tunneling functionality using less bytes per packet for transportation. Such tunneling optimization could be very useful in networks which uses satellite communication links or other means which are not as fast and robust as wired network connection. In such networks extra bytes or packets can have a significant cost on network performance.

Keywords: computer network, ethernet, IP, TUN/TAP, bridge, socket, programming, Unix, Linux, Python, OpenVPN.

AUTOREFERĀTS

Darbā izdevās izstrādāt tīkla tunelēšanas protokolu Zero, kas ir ievērojami efektīvāks par citiem pieejamiem tuneļa risinājumiem, kā piemēram, OpenVPN.

Iztrādāta Zero server datorprogramma GNU/Linux operētājsistēmai, kas realizē darbā aprakstītā Zero protokola darbību praktiski.

Ar eksperimentiem pārbaudīta šī protokola efektivitāte, secināts, ka protokols ir spējīgs 99% no ikdienas sarežģītības tīkla plūsmas tunelēt nepievienojot transportējamai paketei papildus baitus.

SATURS

IEVADS ... **6**

1. ZERO PROTOKOLS ... **8**

1.1. Protokola ideja ... 8

1.2. Transporta veidi ... 11

1.3. Ethernet galveņu sinhronizācija ... 12

1.4. TTL kompensācija ... 13

2. PROTOKOLA REALIZĀCIJA ... **16**

2.1. Programmēšanas līdzekļi ... 16

2.2. Zero serveris ... 18

2.3. Izstrādes vide ... 19

2.4. Zero servera darbība ... 21

2.5. Zero servera uzstādīšana ... 22

3. STATISTIKA ... **27**

3.1. Zero servera statistika ... 27

3.2. Salīdzinājums ar OpenVPN tuneli ... 30

SECINĀJUMI ... **32**

LITERATŪRAS SARAKSTS ... **33**

A. BŪTISKAIS PROGRAMMAS KODS ... **34**

IEVADS

Tīkla tunelēšana ir izplatīts risinājums datortīklu konstruēšanā. Pēc būtības, tas ir tīkla transporta protokols, kurš nodrošina transportējamā tīkla pakešu izolēšanu no tīkla, kuru izmanto tā transportēšanai. To var izmantot, lai transportētu datus cauri citam, nedrošam tīklam, tos kriptējot, vai arī transportēt kādu neatbalstītu protokolu cauri tīklam iekapsulējot to citā, atbalstītā protokolā. Šajā darbā aprakstītais tunelēšanas protokols paredzēts viena tīkla transportēšanai cauri citam tīklam izolējot tos vienu no otra.

Tunelēšana parasti nozīmē katrai paketei pievienot papildus transportēšanas informāciju, kas nepieciešama, lai maršrutētu paketi no vienu datoru uz citu, un otrā galā atgūtu oriģinālo paketi. Piemēram, ja aplūkojam ethernet tunelēšanu, iekapsulējot datus IP/UDP protokolā, šajā gadījumā nepieciešami papildus 42 baiti. Tie sastāv no ethernet, IP, UDP galvenēm – attiecīgi 14, 20 un 8 baiti.

Šie papildus baiti rada vēl lielāku problēmu – tas var kļūt par iemeslu IP paketes fragmentēšanai (sadalīšanai vairākās, mazākās IP paketēs). Tas notiek gadījumā, kad ethernet paketes datu izmērs pārsniedz transporta tīkla MTU (*maximum transmission unit*) [7]. Tādā gadījumā šādas lielas IP paketes vietā ir jāsūta vairākas mazākas.

Šādi gadījumi ir īpaši nelabvēlīgi tīkliem, kur sakari ir lēni, nepietiekami stabili vai līdzīgā veidā ierobežoti. Piemēram, tīkls, kas komunikācijā izmantojot satelītu sakarus, tādā tīklā ir jārēķinās ar nopietniem ierobežojumiem – lēnāki ātrumi, liela aizture, lielāka varbūtība paketi nozaudēt pa ceļam.

Darbā izstrādāts un aprakstīts jauns, eksperimentāls tīkla tunelēšanas protokols Zero, kas ir būtiski efektīvāks par citiem pieejamiem risinājumiem, kā, piemēram, OpenVPN, kurš arī spēj nodrošināt tādu pašu tunelēšanu, bet bez papildus optimizācijas [6].

Zero protokols risina problēmu ar papildus transportēšanas datiem. Ievērojot dažus nosacījumus, tas spēj transportēt IP paketes nepievienojot papildus transportēšanas baitus. Līdz ar to atbrīvojoties no nevēlamas IP pakešu sadalīšanas, jeb fragmentācijas gadījumā, kad ethernet pakete pārsniedz tīkla MTU vērtību. Protokols atbrīvo arī no liekiem baitiem, kas ir vajadzīgi tikai transportam, nevis tuneļa otrajam galam.

Darbs sastāv no vairākām daļām. Nodaļā 1 ir aprakstīta Zero protokola ideja un būtība – pamati, uz ko tas balstās, un kā tas spēj nodrošināt korektu OSI Layer 2 [2] tunelēšanu. Nodaļā 2 ir aprakstīta piedāvātā protokola realizācijas ideja operētājsistēmā Linux, kā arī alternatīvi paņēmieni, ko arī varētu izmantot protokola izstrādē. Nodaļā 3 protokola risinājums tiek pār-

baudīts reālā eksperimentā, nodrošinot tunelēšanas funkciju. Zero salīdzināts ar tāda paša tuneļa tipa nodrošinājumu izmantojot OpenVPN risinājumu. Apkopota statistika par tunelī izsūtītajām paketēm – to skaits, kopējais datu apjoms un tamlīdzīgi. Eksperimentā secināts, ka Zero protokols darbojas ļoti veiksmīgi, tas spēj 99% no tunelējamām IP paketēm pārsūtīt optimizētā veidā.

1. ZERO PROTOKOLS

1.1. Protokola ideja

Lielākā daļa no tīklā transportētām IP paketēm ir nefragmentētas, jo pārsvarā tīkla mezglos ir sabalansētas MTU vērtības. Gandrīz visas IP paketes tiek transportētas izmantojot Ethernet II [8] formātu, kas nosaka standartu MTU 1500. Līdz ar to, relatīvi reti rodas apstākļi, kas piespiestu IP paketi fragmentēt.

IP galvenē ir speciāli paredzēta vieta, kur glabāt informāciju par paketes fragmentāciju (attēls 1). Tā ir MF (*more fragments*) vērtība, kas norāda, vai pakete ir fragmentēta un *fragment offset*, kas nepieciešama, lai varētu korekti atjaunot pilnu IP paketi no fragmentētajām daļām. MF ir viens bits un *fragment offset* ir 13 biti.

Ieviesīsim jēdzienu NICE IP pakete. Tā ir IP pakete ar sekojošiem nosacījumiem:

- MF ir nulle,
- *fragment offset* ir nulle,
- IHL ir 5 (IP galvenes garums ir 20 baiti).

Šādas paketes pakļaujas Zero protokola optimizētai transportēšanai, nepievienojot papildus baitus, visas pārējās IP paketes sauksim par UGLY IP, tās nav iespējams tunelēt optimizētā veidā, tās tiek apstrādātas citādāk. Šīm nefragmentētām NICE IP paketēm varam izmantot 13 *fragment offset* bitus citām vajadzībām. Zero protokols balstās uz šo iespēju izmantot *fragment offset* bitus, kas citādi netiek ņemti vērā nefragmentētām IP paketēm.

Lai tunelētu paketi cauri tīklam, ir nepieciešams maršrutēt to līdz tuneļa otram galam. IP paketei tas nozīmē uzstādīt atbilstošu *destination address* vērtību, kura norāda uz tuneļa otru galu. Tuneļa otrajā galā ir jāiegūst oriģinālā pakete, tas nozīmē atjaunot šo *destination address* vērtību uz tādu, kas bija paketei pirms nonākšanas tunelī. Ir nepieciešams šo oriģinālo adreses vērtību kaut kādā veidā aizsūtīt līdz otrajam galam. Pilns adreses garums ir 4 baiti, jeb 32 biti, bet, iepriekš minēts, ka ir pieejami tikai 13 *fragment offset* biti, kurus varētu izmantot. Nevaram saglabāt visu adresi, bet tikai 13 bitus no tās.

Ar to pietiek, lai nodrošinātu maršrutizāciju, bet tas gan uzliek transporta tīklam zināmu ierobežojumu. Ideja ir maršrutēt pēc *destination address* pirmajiem 8 bitiem (pirmo baitu). Izmantojam 8 no pieejamiem 13 *frame offset* bitiem, lai saglabātu IP paketes oriģinālo pirmo baitu. Atlikušie 5 biti tiek atstāti ethernet galvenes transportam, kas ir aprakstīts nodaļā 1.3.

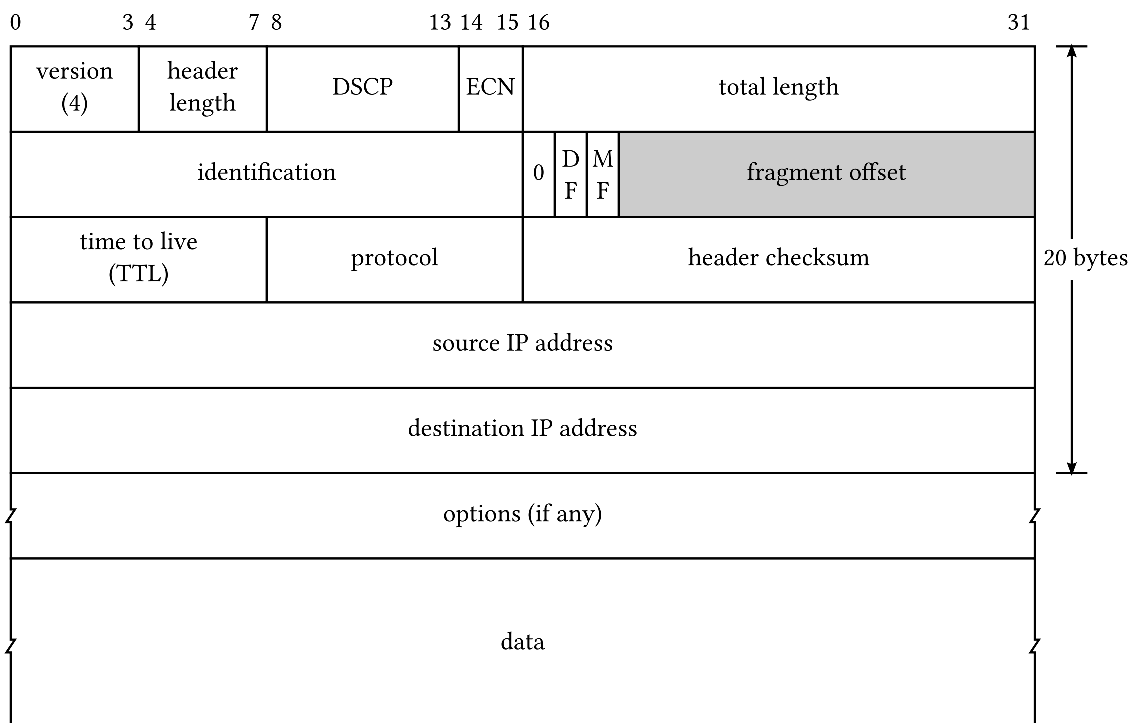
Adresācija pēc pirmā baita nozīmē maršrutēt /8 tīklu – tas ir transporta tīkla ierobežojums – transporta tīklam ir jāspēj nodrošināt šī maršrutēšana.

Katram tuneļa galam nepieciešams piešķirt savu /8 tīkla adresi (skaitli no 0 līdz 255). Piemēram, tuneļa A puse varētu būt 88.0.0.0/8, bet B 77.0.0.0/8. Tuneļa transporta tīklam ir nepieciešams uzstādīt maršrutēšanas likumus, kas novadītu IP paketes ar *destination address* 88.a.b.c un 77.d.e.f attiecīgi A un B tuneļa galiem.

Algoritms ir sekojošs. Tunelī ienākošai paketei pirmo baitu no IP *destination address* saglabā brīvajos 13 *frame offset* bitos. Nomaina *destination address* pirmo baitu uz tādu, kas nodrošina maršrutēšanu līdz tuneļa otrajam galam, piemēram, vērtība 77, attiecīgi maršrutēšana pēc 77.0.0.0/8 (maršrutē virzienā $A \to B$). Tuneļa otrajā galā, saņemot paketi, šis *destination address* pirmais baits tādā pašā veidā tiek nomainīts atpakaļ uz oriģinālo, kas glabājas *frame offset* laukā, un tiek iegūta oriģinālā IP pakete.

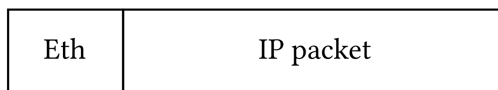
Līdz šim aprakstītais algoritms nodrošina OSI Layer 3 tunelēšanu, bet Zero piedāvā OSI Layer 2. Tas nozīmē, ka līdz otrajam galam ir jātransportē arī ethernet daļa (ethernet galvene).

Varam runāt par NICE ethernet paketi – tā ir pilna ethernet pakete, kas sastāv no divām daļām – 1) ethernet galvenes daļa, 2) NICE IP pakete (attēls 2). Tātad, līdz otrajam galam ir jāaizsūta arī ethernet galvene. Šim nolūkam tiek izmantoti atlikušie 5 biti no *fragment offset*

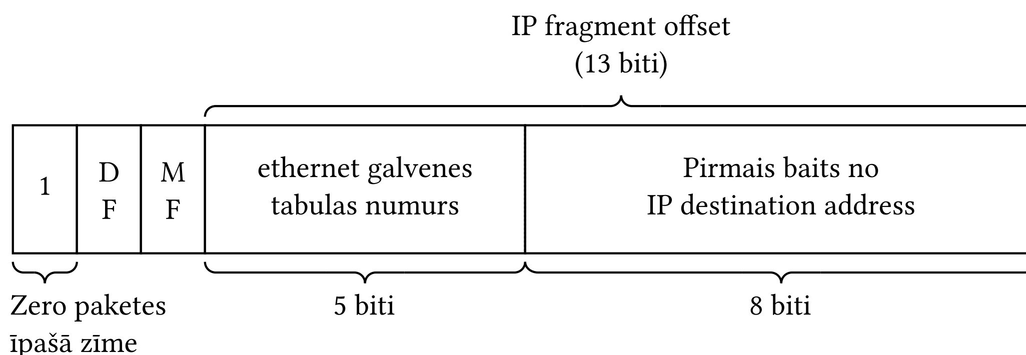


Att. 1: IP galvene.

lauka. Tas ļauj saglabāt skaitli no 0 līdz 31. Tādā veidā tiek izmantoti visi brīvie *frame offset* biti kā parādīts attēlā 3.



Att. 2: NICE pakete.



Att. 3: NICE IP *fragment offset* izmantošana.

Risinājums ir "atcerēties" sastaptās ethernet galvenes un piešķirt katrai skaitli no 0 līdz 31, tādā veidā unikāli identificējot tās. To ir vērts darīt, jo ethernet galvenes ir relatīvi nemainīgas no paketes uz paketi, kā arī nav pārāk daudz variāciju vienas mašīnas ietvaros. Minimāli ethernet galvenē parasti parādās "kaimiņu" tīkla mezglu MAC adreses, protokola identifikators – vērtības, kas nav ļoti mainīgas. Tajā skaitā statiski ir arī VLAN (*virtual local area network* [9]), MPLS (*Multiprotocol Label Switching* [10]) un līdzīgi ethernet līmeņa protokolu identifikatori – arī tos varam tādā pašā veidā "atcerēties". Ieviesīsim jēdzienu ZERO IP pakete – tā ir IP pakete, kas ir pārveidota aprakstītajā veidā:

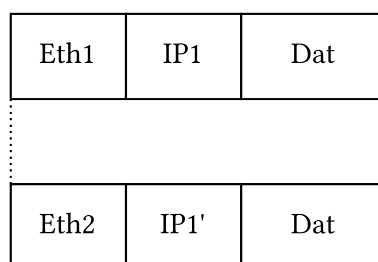
- *frame offset* pēdējos 8 bitos glabājas oriģinālais *destination address* pirmais baits,
- *frame offset* pirmajos 5 bitos ir tabulas skaitlis, kas atbilst oriģinālajai ethernet galvenei,
- īpaša iezīme neizmantotajā IP laukā (*evil bit* [11]), kas norāda, ka šī ir ZERO IP pakete.

Pievienojot iepriekš aprakstīto ideju, Zero algoritms kļūst sekojošs. Ienākošajai paketei nomaina *destination address* pirmo baitu un oriģinālo ievieto 8 no 13 pieejamajiem *frame offset* bitiem. Identificē ethernet galveni, atrod tai atbilstošo skaitli no 0 līdz 31, un ievieto to atlikušajos 5 *frame offset* bitos. Šādu ZERO IP paketi ievada transporta tīklā. Tā tiek maršrutēta pēc *destination address* pirmā baita. Nonākot otrajā galā, tiek atjaunots *destination address*

pirmais baits no saglabātā oriģinālā baita, kas glabājas 8 *frame offset* bitos. Tiek atrasta atbilstošā ethernet galvene no vērtības, kas ir saglabāta atlikušajos *frame offset* 5 bitos. Tādā veidā tiek sastādīta atbilstoša ethernet pakete, kas ir gandrīz (nodaļa 1.4) identiska oriģinālajai, kas ienāca tunelī. Atjaunotā ethernet pakete tiek ievadīta tīklā, izkļūstot ārā no tuneļa.

1.2. Transporta veidi

Lai nodrošinātu Zero protokola darbību, ir nepieciešami vairāki transporta veidi, kā pakete tie virzīta cauri tunelim. Viens no to veidiem ir "veiksmīgais" varianats – ZERO IP transports (attēls 4), šajā gadījumā nodrošinām tunelēšanu izmantojot aprakstīto IP paketes manipulāciju, izvairoties no datu pievienošanas. Transportējamai paketei garumā x baiti, vajadzīgi arī tikpat x baiti, lai to nogādātu tuneļa otrā galā.



Att. 4: NICE ethernet pakešu tunelēšana.

Tomēr ir gadījumi, kad šo transportu nevar izmantot. Piemēram, ja IP pakete ir UGLY IP, tādā gadījumā nav iespējama aprakstītā IP manipulācija, un paketi līdz otrajam galam ir jātransportē citā veidā. Līdzīgi ir situācijā, kad līdz tuneļa otrajam galam ir jāaizsūta informācija par ethernet galvenes atbilstošo skaitli. Kad pirmo reizi tunelī ienāk NICE pakete ar ethernet daļu E un izvēlēts atbilstošs skaitlis e, tad to ir nepieciešams paziņot tuneļa otrajam galam. Tādā gadījumā nevaram izmantot ZERO IP transportu (ethernet tabulu sinhronizācija ir aprakstīta nodaļā 1.3). Šādos gadījumos tiek sūtīta pilna ethernet pakete ierakstīta UDP protokolā (attēls 5). Turpmāk sauksim šo transporta metodi par ZERO UDP.

Attēlā 5 ir redzams, ka šī metode pievieno papildus datus. Faktiski tā ir vienkārša ethernet pakešu tunelēšana izmantojot UDP protokolu. Varētu teikt, ka pakete tiek "apaudzēta" ar papildus ethernet, IP, UDP galvenēm, kas attiecīgi ir 14, 20 un 8 baitus garas. Papildus nepieciešama minimāla Zero galvene (attēlā 5 apzīmēta ar Z), kas nodod otrajai pusei papildus informāciju par transportēto ethernet paketi.

Tā varētu būt situācija, kad ir jātransportē UGLY IP. Tad Z, kas šajā gadījumā ir viens

baits, tiek uzstādīta vērtība viens. Šādā reizē iekapsulētā ethernet pakete tiek ievadīta tīklā to papildus neapstrādājot. Otrs gadījums ir ziņojums par jaunu ethernet galveni un tās atbilstošo skaitli. Tādā gadījumā Z ir divu baitu garš – pirmais baits ir 0, kas norāda šo gadījumu, otrs baits ir ethernet galveņu tabulas skaitlis, kas atbilst šīs paketes ethernet daļai.

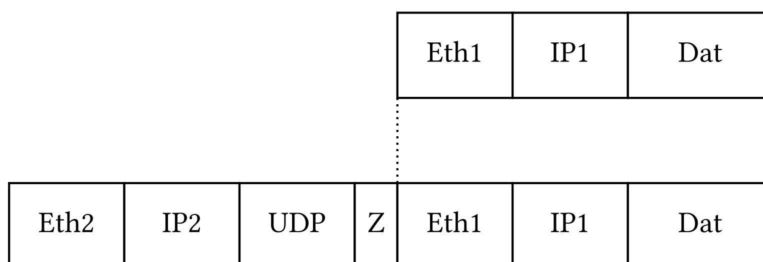
1.3. Ethernet galveņu sinhronizācija

Otrajam galam, saņemot ZERO IP paketi, ir jāatgūst oriģinālā ethernet pakete, kas nonāca tunelī. Atjaunot IP *destination address* ir vienkārši, izmainītais baits glabājas *frame offset* laukā. Sarežģītāk ir atjaunot oriģinālo ethernet galveni, jo šajā gadījumā ir tikai tai atbilstošs skaitlis no 0 līdz 31.

Tuneļa katrā galā tiek glabāta informācija par sastaptajām ethernet galvenēm. Tā ir tabula ar 32 rindām, kur skaitlim no 0 līdz 31 atbilst ethernet galvene – zinot skaitli, varam dabūt tam atbilstošo ethernet galvenes baitu virkni.

Ir izdevīgi ieviest katrā galā divas tādas tabulas, viena virzienam $A \to B$, otra virzienam $B \to A$. Katrai šādai tabulai viens tuneļa gals saglabā ienākošo ethenet galveņu ierakstus, otrs gals tikai saņem informāciju par šiem ierakstiem. Tā tabulā $A \to B$ jaunus ierakstus veido tuneļa A gals – tās ir ethernet galvenes paketēm, kas ienāk tunelī no A puses. Analoģiski tiek veidoti ieraksti $B \to A$ tabulai, tajā tuneļa gals B veido jaunus ierakstus un A puse tikai saņem informāciju par tiem.

Algoritms ir sekojošs. Ienākot jaunai NICE ethernet paketei tuneļa A pusē, tiek samekLēts paketes ethernet daļas atbilstošais skaitlis. Ja tabulā neeksistē ieraksts ar šo ethernet galveni, tad tāds ir jāizveido. Galvene ir jāsaglabā tabulā $A \to B$ ar brīvu skaitli n un jāsūta pilna ethernet pakete B pusei, paziņojot tai, izmantojot ZERO UDP transporta veidu (1.2), ka tika saglabāta jauna ethernet galvene ar skaitli n. B puse šo paketes ethernet daļu saglabā savā $A \to B$ tabulā ar tieši šo skaitli n.



Att. 5: UGLY ethernet pakešu tunelēšana.

Papildus drošībai nepieciešams ieviest atkārtotu ethernet galveņu skaitļu pārsūtīšanu. Piemēram, gadījumā, kad jaunu ethernet galveņu sūtīšanas brīdī, tīkls nav bijis pieejams, A puse nevar uzzināt, ka šīs paketes nav nokļuvušas līdz B, jo protokols ir paredzēts darboties arī vienā virzienā, līdz ar to, nav iespējams ieviest apstiprinājumu ziņojumus.

Saņemot paketi un atrodot tai atbilstošo ethernet galvenes tabulas ierakstu, nepieciešams uzzināt, pirms cik ilga laika šis ieraksts tika nosūtīts B pusei. Ja iepriekšējais pilnais sūtījums attiecīgajai ethernet daļai ir bijis vairāk kā pirms 10 sekundēm, tad šī pakete ir jāsūta atkārtoti ar ZERO UDP transporta metodi, ziņojot par šo ethernet tabulas ierakstu. Šādā veidā var nodrošināt sinhronizācijas atjaunošanos, ja kaut kas ir noticis ar tīklu, vai kādu no tuneļa programmām.

Piemēram, ja B puse tiek pārstartēta, tad tai tabulas ir tukšas, bet A puse to nevar uzzināt. A vēl joprojām, saņemot ethernet paketes, tās transportē ZERO IP formā uzskatot, ka B pusei ir atbilstošs ethernet galveņu ieraksts. Ieviešot šādu ethernet tabulas ierakstu pārsūtīšanas veidu, tiek nodrošināta sinhronizācijas atjaunošana. Sliktākais scenārijā, kad B pusei nav atbilstošā ethernet galveņu ieraksta, ir 10 sekunžu intervāls, kurā paketes netiek cauri tunelim, jo vēl nav nodrošināta sinhronizācija. Pēc 10 sekundēm tabulas ieraksti ir jau atjaunoti, un protokols spēj nodrošināt normālu transportu.

Tā kā ethernet galveņu tabulai var būt tikai 32 rindas, bet ethernet daļu veidi varētu būt vairāk nekā 32, tad ir nepieciešams pārvaldīt šos ierakstus veidā, kas ļauj atbrīvoties no veciem, nevajadzīgiem ierakstiem un dod vietu jauniem. Piedāvātais risinājums ir reizi 10 minūtēs iztīrīt tabulu, izmest 10 vissenāk izmantotos ierakstus.

1.4. TTL kompensācija

TTL vērtība ir viena baita IP protokola lauks, ar maksimālo vērtību 255. IP protokols paredz TTL vērtības izmantošanu sekojošā veidā. IP paketei, tiekot transportētai tīklā, katrā maršrutēšanas solī TTL vērtība tiek samazināta par viens. Tas tika ieviests, lai ierobežotu IP paketes dzīves ilgumu, kas ļauj izvairīties no bezgalīgiem pakešu ceļošanas cikliem. Kad IP paketes TTL sasniedz nulli, tāda pakete tiek izmesta. Ja TTL ir 255, tas atļauj paketei būt maršrutētai 255 reizes pirms tā tiek izmesta.

Visbiežāk IP komunikācijā TTL vērtību mēdz uzstādīt 64. Bet ir gadījumi, kad izmanto speciāli sagatavotas IP paketes ar mazākām TTL vērtībām, piemēram, `traceroute` programma. Tā sūta speciāli sagatavotas IP paketes ar mazām TTL vērtībām, sagaidot ICMP kļūdas atbildi

par TTL nulles vērtību, tādā veidā apzinot tīklu.

Līdz šim aprakstītajam protokolam ir sekojoša problēma, kas saistās ar IP TTL vērtību. Transportējot NICE IP paketes, modificējot tunelī ienākošo IP paketi, pēc būtības tā pati transportējamā IP pakete tiek izmantota arī transportam. No tā seko, ka transportējamās IP paketes TTL vērtība tiek samazināta transportējot to cauri tīklam. Tuneļa otrajā pusē atjaunojot *destination address* un ethernet galveni, vajag atjaunot arī TTL vērtību, jo pretējā gadījumā neiegūstam identisku paketi tuneļa otrā galā.

Jāpiemin, ka parastai TCP/IP komunikācijai, ko izmanto, piemēram, tīmekļa pārlūkošanai un citiem ikdienas darbiem, netraucē šī TTL vērtības samazināšana. Bet tādu specifisku programmu, kā `traceroute` uzvedība nebūs korekta.

Aprakstītajam Zero protokolam ir jāpievieno metode, kas nodrošina korektu OSI Layer 2 tunelēšanu, nodrošinot paketes identiskumu. Nepieciešams papildus nodrošināt sekojošo:

1. IP paketes ar mazu TTL vērtību nokļūst tuneļa otrā galā,

2. TTL vērtība tunelī ieejot ir vienāda ar vērtību izejot no tā.

Maršrutēšanas soļu skaits starp divām tīkla iekārtām ir relatīvi nemainīgs. Tas var mainīties, ja, piemēram, tīklā kādas maršrutēšanas iekārtas uzstādījumi tiek pamainīti, mainot pakešu plūsmas ceļu. Vai arī kad tīklā kāds maršrutētājs tiek izslēgts, tad mainās paketes transportēšanas ceļš, līdz ar to TTL vērtības samazinājums varētu būt cits.

Zero ir jānodrošina TTL vērtības kompensācija. Vienkāršākais veids, kā to izdarīt, ir ieviest konfigurējamu parametru `ttl_delta`, kas tiek pieskaitīts paketes TTL vērtībai tādā veidā to kompensējot un nodrošinot paketes identiskumu.

Tomēr tādai metodei ir zināmas problēmas. Pirmkārt, ir jānoskaidro reālais maršrutēšanas soļu skaits, reāls lielums, par kādu tiek samazināta TTL vērtība tunelī. Tas ievieš papildus korektas uzstādīšanas sarežģītību. Otrkārt, kad transportēšanas tīklā tiek veiktas kādas izmaiņas, kas maina šo soļu skaitu, tad arī ir attiecīgi jāmaina Zero `ttl_delta` vērtība.

Labāks risinājums ir iegūt `ttl_delta` vērtību dinamiski, programmai izpētot reālo transporta tīklu. Zero protokols paredz ik pēc 10 sekundēm izsūtīt speciāli sagatavotu IP paketi ar ZERO IP transporta metodi, kas ļauj otrajai pusei uzzināt reālo `ttl_delta`.

Tuneļa gals A nosūta B pusei speciāli sagatavotu IP paketi ar TTL vērtību 255. B puse, saņemot šadu paketi, identificē to kā speciālu TTL uzzināšanas paketi, attiecīgi nolasa šīs saņemtās paketes TTL vērtību. Šī nolasītā TTL vērtība B pusē ir samazināta transportējot

to cauri tīklam, tādā veidā varam uzzināt šo samazinājumu. Atņemot no 255 šo TTL vērtību, iegūstam reālo `ttl_delta`.

Šāda TTL kompensācija ir jāveic paketei izejot no tuneļa. Respektīvi, ja virziens ir no A uz B, tad to kompensē B pusē. Nav iespējams iepriekš kompensēt TTL vērtību jau A pusē, jo:

1. ja paketei jau ir TTL 255, tad to vairs nevar palielināt,
2. A puse nevar zināt virziena $A \to B$ maršrutēšanas soļu skaitu, jo protokols ir paredzēts arī vienvirziena darbībai (nevar paļauties uz informācijas saņemšanu no B).

Lai nodrošinātu tādu IP pakešu izkļūšanu cauri tunelim, kurām ir maza TTL vērtība, ir jāievēro vēl viens nosacījums. Cauri tunelim izkļūst tikai tādas IP paketes, kurām TTL ir lielāks par maršrutēšanas soļu skaitu, jo citādāk TTL sasniegs nulles vērtību, un pakete tiks izmesta. Līdz ar to, nevaram tunelēt ar ZERO IP transportu IP paketes, kurām TTL ir mazāka par `ttl_delta`.

Šī problēma tiek risināta sekojošā veidā. IP paketes, kurām TTL ir mazāks par `ttl_delta` tiek transportētas, ar ZERO UDP transporta metodi (attēls 5, lpp. 12) – sūtot pilnas ethernet paketes.

Jāņem vērā apstāklis, ja komunikācija ir iespējama tikai virzienā $A \to B$, tad ir jāpaļaujas uz uzstādījumu A pusē, kas nosaka minimālo TTL vērtību IP paketei, ko var sūtīt ar ZERO IP metodi. Tas seko no vienvirziena komunikācijas – A gals nevar saņemt B puses reālo `ttl_delta`. Identitātes problēma otrajā galā nebūs, jo B zinās reālo TTL vērtību, ko atjaunot.

2. PROTOKOLA REALIZĀCIJA

2.1. Programmēšanas līdzekļi

Protokola realizācija paredzēta GNU/Linux operētājsistēmai, tā ir izstrādāta un testēta tieši šajā sistēmā. Bet tajā nav specifiski risinājumi, kas neļautu to darbināt, iespējams ar nelieliem labojumiem, arī citās operētājsistēmās.

Linux operētājsistēma piedāvā vairākus veidus, kā programmēt tīkla risinājumus ethernet pakešu apstrādes līmenī – saņemot, modificējot un ievadot atpakaļ tīklā pilnas ethernet paketes:

- Linux kodola programmēšana,
- TUN/TAP iekārtas,
- *raw socket*.

Katrai no metodēm ir labās un sliktās īpašības.

Raw socket (*datalink access* līmeņa saskarne [12]) ļauj nolasīt visas paketes, kas ienāk sistēmā, kā arī ievadīt tīklā patvaļīgi sagatavotas datu virknes (ethernet paketes). *Raw socket* tipiski izmanto tādas programmas, kā `ping`, `traceroute`, `tcpdump` un citas. *Raw socket* ir pieejams vairākās operētājsistēmās, tomēr to saskarne un specifika var ļoti atšķirties, it īpaši darbības ar OSI layer 2 (*datalink access*) slāni [1].

Linux kodola programmēšana ir pats "zemākais" līmenis Linux sistēmā, kurā var darboties ar IP un Ethernet kadriem, tos saņemt, apstrādāt un sūtīt. Šajā līmenī, programmētājam ir visas iespējas realizēt jebkuru scenāriju. Tomēr izmaiņas kodolā nav ērtākais veids, kā pievienot sistēmai jaunu vai mainīt esošu funkcionalitāti, vēl mazāk, eksperimentēt. Pie tam funkcionalitāte balstīta šajā līmenī ir specifiska Linux sistēmai [13].

Daudzas modernas operētājsistēmas, tajā skaitā Linux, piedāvā veidot TUN/TAP tīkla iekārtas. Tās ir virtuālas tīkla iekārtas, kas darbojas tādā pašā veidā, kā īstās – tās var konfigurēt līdzīgi kā īstās iekārtas. Šīs TUN/TAP iekārtas piedāvā iespēju procesiem pieslēgties pie tām, lasīt tajā ienākošās un attiecīgi arī rakstīt IP un ethernet paketes. TAP iekārtas nodrošina darbību OSI Layer 2 līmenī, TUN OSI Layer 3 līmenī [2]. Ievadītās paketes operētājsistēma apstrādā tādā paša veidā, kā parastas tīkla iekārtas gadījumā [4]. TUN/TAP iekārtas ir realizētas tādās populārās operētājsistēmās, kā:

- Linux,
- FreeBSD,
- Solaris,
- Microsoft Windows.
- Mac OS X.

Protokolu izlemts realizēt ar virtuālajām TUN/TAP iekārtām, jo šajā līmenī ir mazāka atšķirība starp operētājsistēmām – ir daudz vieglāk programmu, kas paredzēta Linux sistēmai, pārtaisīt tā, lai tā darbojas arī, piemēram, BSD operētājsistēmā.

TUN/TAP izvēli pastiprina tas, ka arī daudzu citu tunelēšanas protokola programmatūru realizācijas ir balstītas uz šīm iekārtām. TUN/TAP realizācijas variantu izmanto arī tādas tunelēšanas programmas, kā, piemēram,

- OpenVPN,
- VTun,
- OpenSSH.

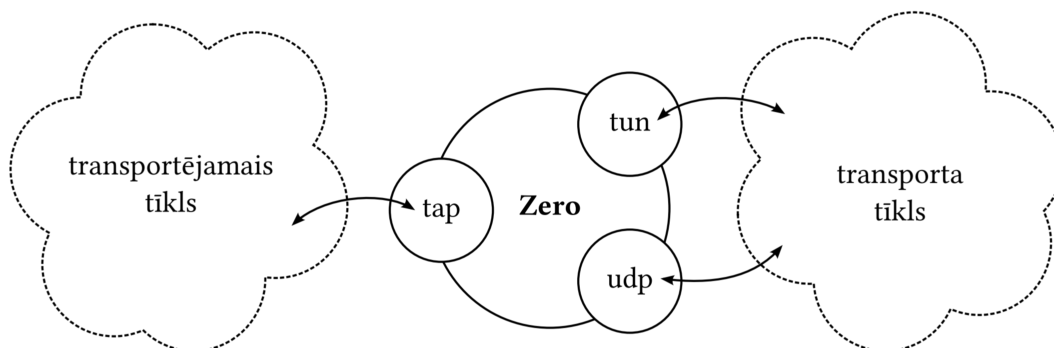
Tā kā protokols ir pilnīgi jauns un eksperimentāls, tas vēl var daudz mainīties. Ieviešot to reālā darbībā, varētu būt saskare ar problēmām, par kurām iepriekš nav padomāts. Līdz ar to, tika izvēlēts pirmajai realizācijai izmantot programmēšanas valodu Python, kas ļoti labi ir piemērota prototipēšanai [5].

> In The Mythical Man-Month, Fredrick Brooks suggests the following rule when planning software projects: "Plan to throw one away; you will anyway." Brooks is saying that the first attempt at a software design often turns out to be wrong; unless the problem is very simple or you're an extremely good designer, you'll find that new requirements and features become apparent once development has actually started.

Python neuzspiež programmēt izmantojot kādu konkrētu programmēšanas paradigmu, tajā var programmēt gan objektorientēti, procedurāli, imperatīvi, arī funkcionāli. Līdz ar to, ir iespējams izvēlēties tādus programmēšanas līdzekļus, kas viegli tulkojas citā valodā. Sākotnējā realizācija rakstīta ar domu, ka tā varētu tikt pārrakstīta programmēšanas valodā C, kas uzlabo programmas ātrdarbību.

2.2. Zero serveris

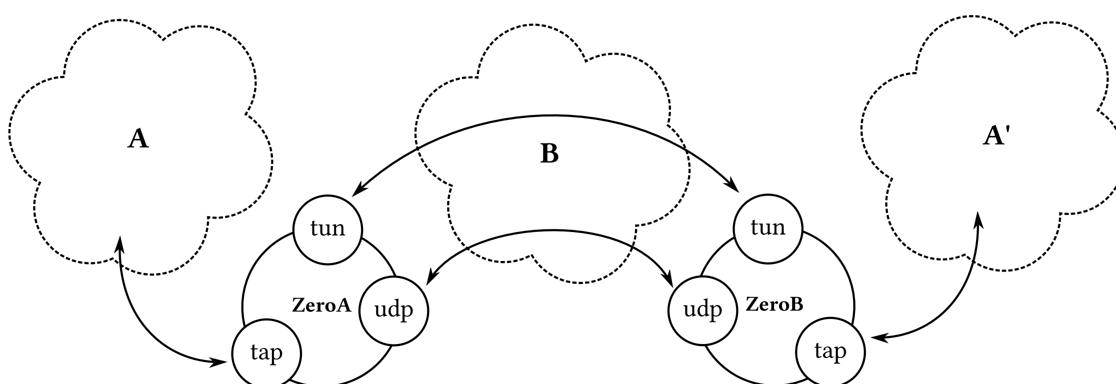
Zero protokolu nodrošina servera process, kas klausās TUN/TAP iekārtas, lasa no tām paketes, apstrādā tās atbilstoši Zero protokola algoritmam (nodaļa 1) un ievada tās tālāk tīklā. Attēlā 6 ir shematiski parādīti Zero servera procesa datu ievades un izvades kanāli – procesa saskares punkti ar "ārpasauli".



Att. 6: Zero servera pakešu ievades, izvades punkti.

Zero serverim var izdalīt trīs dažādus datu ievades un izvades punktus. Savienojums ar TAP iekārtu nodrošina tuneļa ieeju un izeju – tas ir saskares punkts ar "ārējo vidi". Šeit tiek nolasītas ethernet paketes, kuras ir jāaizgādā līdz tuneļa otrajam galam. Šis pats punkts tiek izmantots arī no otra gala saņemto pakešu izvadei ārā no tuneļa.

TUN savienojums tiek izmantots, lai saņemtu un sūtītu modificētās ZERO IP paketes (nodaļa 1.2). TUN saskarne nodrošina OSI Layer 3 piekļuvi paketei, šādā veidā varam saņemt un sūtīt tieši IP paketes, izvairoties no ethernet līmeņa.



Att. 7: Tuneļa abu galu komunikācijas punkti.

Trešais komunikācijas punkts ir parasta UDP *socket* saskarne, kuru izmanto ZERO UDP transportam. Kā jau iepriekš aprakstīts (nodaļa 1.2), to izmanto, lai transportētu UGLY IP

paketes, kā arī jaunu ethernet galveņu paziņojumus. Zero serveris sūta otrajam galam, kā arī klausās UDP portu un saņem no otra tuneļa gala sūtītās ZERO UDP paketes.

Lai nodrošinātu Zero tuneli, abos galos ir jābūt uzstādītam Zero serverim, kas nodarbojas ar pakešu lasīšanu, modificēšanu, rakstīšanu atbilstoši Zero protokolam. Attēlā 7 ir shematiski attēlota situācija, kur divi Zero serveri nodrošina tuneli starp A un A' tīkla daļām (tuneļa pusēm), tunelējot paketes cauri tīklam B.

Nodaļā 1.1 aprakstīts, ka Zero protokols ir atkarīgs no maršrutēšanas pēc /8 tīkliem. Lai protokols darbotos, ir nepieciešams iesaistītajās sistēmās veikt uzstādījumus, kas nodrošina modificēto ZERO IP pakešu nokļūšanu līdz otrajam tuneļa galam. Jāizveido maršrutizēšanas likumi izvēlētajiem /8 tīkliem. Jāsavieno nepieciešamās iekārtas ar *bridge*. Jāuzstāda vajadzīgie IP *forwarding* noteikumi. Nodaļā 2.3 aprakstīts Zero lietošanas gadījums ar tam nepieciešamajiem uzstādījumiem.

2.3. Izstrādes vide

Strādājot pie Zero protokola realizācijas, tika izmantota virtualizācijas vide VirtualBox, lai veidotu virtuālās mašīnas, kā arī konstruētu dažādus tīklus. Tika izveidota datoru un tīklu konfigurācija, kura pēc iespējas precīzāk atbilst reālai vide, kurā varētu būt lietojams Zero serveris.

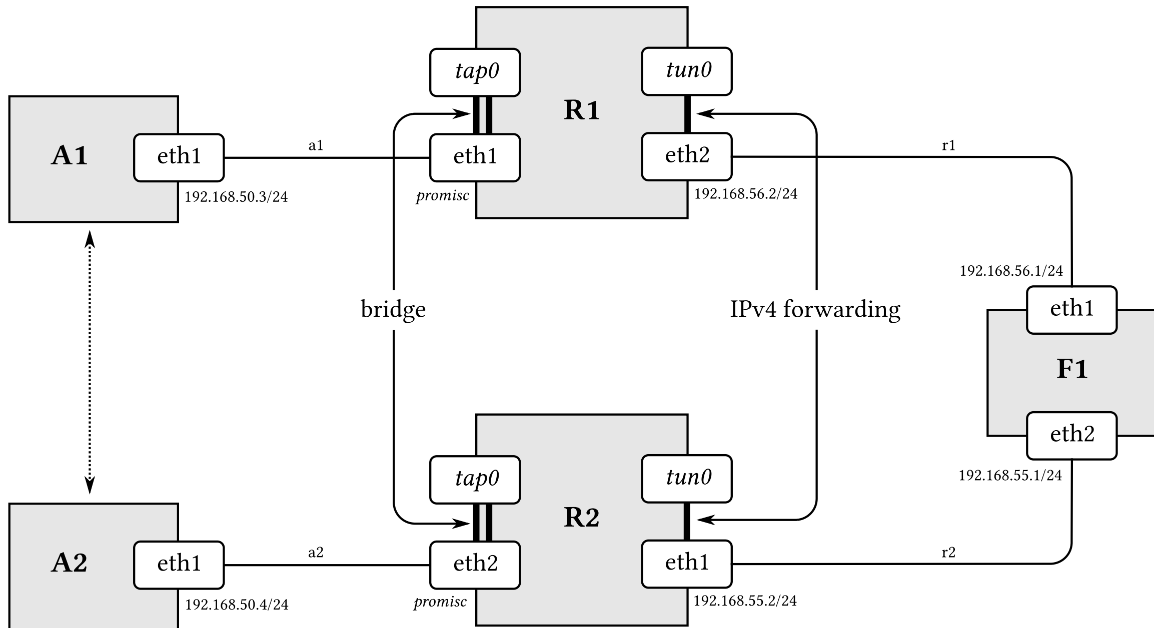
Tika simulēta situācija, kur Zero tunelēšanas protokolu nodrošina divas atsevišķas mašīnas ($R1$ un $R2$), katra ar divām tīkla iekārtām. Tādas, ar Zero serveri aprīkotas mašīnas, varētu izmantot, lai savienotu tīklus ar Zero tuneļa palīdzību. Šādas iekārtas varētu "iespraust" tīkla vajadzīgajās vietās, veikt nepieciešamos maršrutēšanas uzstādījumus, un tunelis ir nodrošināts.

Attēlā 8 ir redzama virtuālās vides konfigurācija, kas tika veidota, lai varētu pārbaudīt Zero darbību simulētā vidē. $A1$ un $A2$ datori ir tuneļa lietotāji, tie ir patvaļīgi transportējamā tīkla dalībnieki, kuri neko "nezina" par tuneli. $R1$ un $R2$ ir datori, kas nodrošina tunelēšanu, tajos darbojas Zero serveris. $F1$ iekārta simulē situāciju, kur transporta tīkls sastāv no vairākām daļām, reāli tie varētu būt daudzi datori, cauri kuriem vajadzētu maršrutēt tuneļa paketes. $F1$ ieviests, lai pārbaudītu darbībā nepieciešamos maršrutēšanas noteikumus.

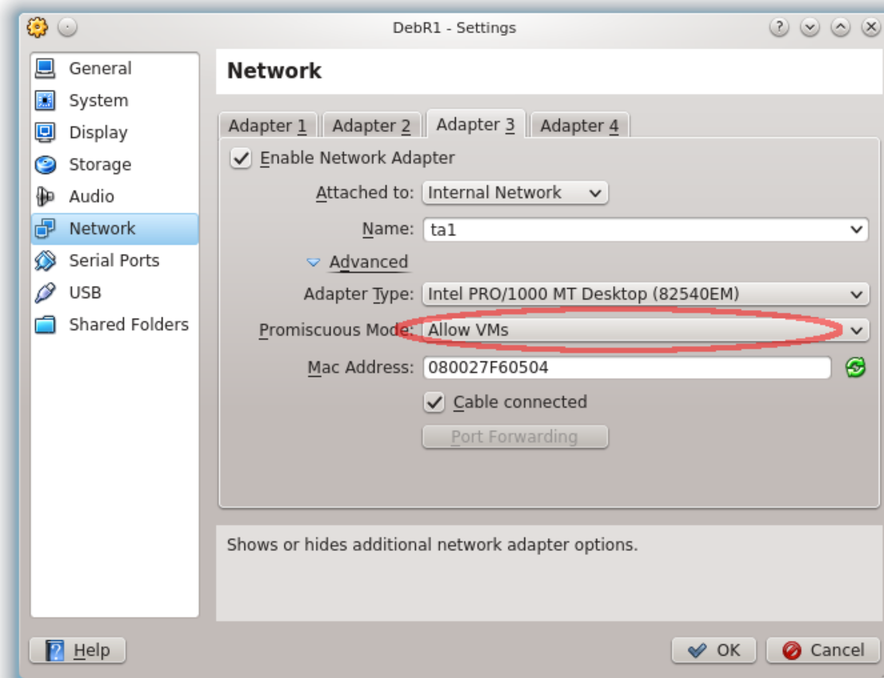
$R1$ un $R2$ datoru attiecīgi $eth1$ un $eth2$ tīkla iekārtas darbojas *promiscuous* režīmā, tās saņem visas paketes, kas nāk no $A1$, $A2$. Tas ļauj izvairīties no šo ierīču adrešu uzstādīšanas. Jāpiezīmē, ka, lai VirtualBox vidē ieslēgtu *promiscuous* funkcionalitāti, nepietiek tikai ar to,

ka pašās virtuālajās operētājsistēmās tīkla iekārtām uzstāda šo režīmu. Ir nepieciešams arī VirtualBox uzstādījumos to atzīmēt, kā redzams attēlā 9.

Lai paketes nokļūtu tunelī, kā arī varētu no tā izkļūt, nepieciešams savienot Zero izmantotās TAP iekārtas ar attiecīgajām tīkla ierīcēm. $R1$ ir nepieciešams savienojums $eth1$ un



Att. 8: Izstrādes un eksperimentu vide.



Att. 9: VirtualBox *promiscuous* režīma uzstādīšana.

$tap0$, $R2$ $eth2$ un $tap0$. Līdzīgi ir ar tuneļa ZERO IP transporta mehānismu. Lai paketes varētu nokļūt no TUN iekārtas uz īstās tīkla ierīces, nepieciešams uzstādīt IP *forwarding*. $R1$ datorā starp iekārtām $tun0$ un $eth2$, $R2$ datorā starp $tun0$ un $eth1$.

Katram Zero tuneļa galam atvēlēts savs /8 tīkls. Eksperimentā izmantojam sekojošus tīklus: 77.0.0.0/8 un 88.0.0.0/8. Pirmo maršrutē uz Zero servera TUN savienojumu datorā $R2$ ($R2$, $tun0$), tīkls Otro maršrutē uz TUN savienojumu datorā $R1$ ($R1$, $tun0$). Ir nepieciešami sekojoši maršrutēšanas likumi:

Datorā $R1$:

```
# route add -net 77.0.0.0 netmask 255.0.0.0 gw 192.168.56.1
# route add -net 88.0.0.0 netmask 255.0.0.0 dev tun0
# route add -net 192.168.55.2 netmask 255.255.255.255 gw 192.168.56.1
```

Datorā $F1$:

```
# route add -net 77.0.0.0 netmask 255.0.0.0 gw 192.168.55.2
# route add -net 88.0.0.0 netmask 255.0.0.0 gw 192.168.56.2
```

Datorā $R2$:

```
# route add -net 77.0.0.0 netmask 255.0.0.0 dev tun0
# route add -net 88.0.0.0 netmask 255.0.0.0 gw 192.168.55.1
# route add -net 192.168.56.2 netmask 255.255.255.255 gw 192.168.55.1
```

2.4. Zero servera darbība

Attēlā 10 ir detalizēti parādīta Zero servera darbība. Ar soļiem tiek aprakstīts, kas notiek ar paketi, kas tiek izsūtīta no $A1$ datoram $A2$, kādā veidā tā tiek cauri tunelim.

1. Ethernet pakete izsūtīta no $A1$ $eth1$ tīkla iekārtas nonāk $R1$ $eth1$ iekārtā, jo fizisks savienojums, un $R1$ $eth1$ ir *promiscuous* režīmā.

2. $R1$ $eth1$ iekārta ir saistīta ar $tap0$ izmantojot *bridge*, līdz ar to, ienākošā ethernet pakete nokļūst $tap0$.

3. Zero serveris nolasa $tap0$ iekārtā ienākošo ethernet paketi.

3a. Ja pakete ir NICE IP, tad tā tiek apstrādāta un ierakstīta $tun0$ iekārtā. Tai tiek nomainīts IP *destination address* pirmais baits uz 77. Pakete tiek pārveidota par ZERO IP.

3a'. Starp $tun0$ un $eth2$ ir uzstādīts IP *forwarding* un attiecīgi maršrutēšanas noteikumi. Pakete nonāk $eth2$ iekārtā.

3b. Ja pakete ir UGLY IP, tā to nosūta ar ZERO UDP transportu un ziņojumu, ka tā ir UGLY. Pakete nonāk $eth2$ iekārtā.

4. Pakete nokļūst līdz nākamajai iekārtai.

5. Paketi padod tālāk $R2$ iekārtai.

6a. ZERO IP pakete tiek novadīta $tun0$ ierīcē, jo ir IP *forwarding* un attiecīgi maršrutēšanas noteikumi.

6a' Zero serveris nolasa $tun0$ iekārtā ienākošo paketi.

6b. ZERO UDP transporta sūtījums nonāk līdz Zero serverim.

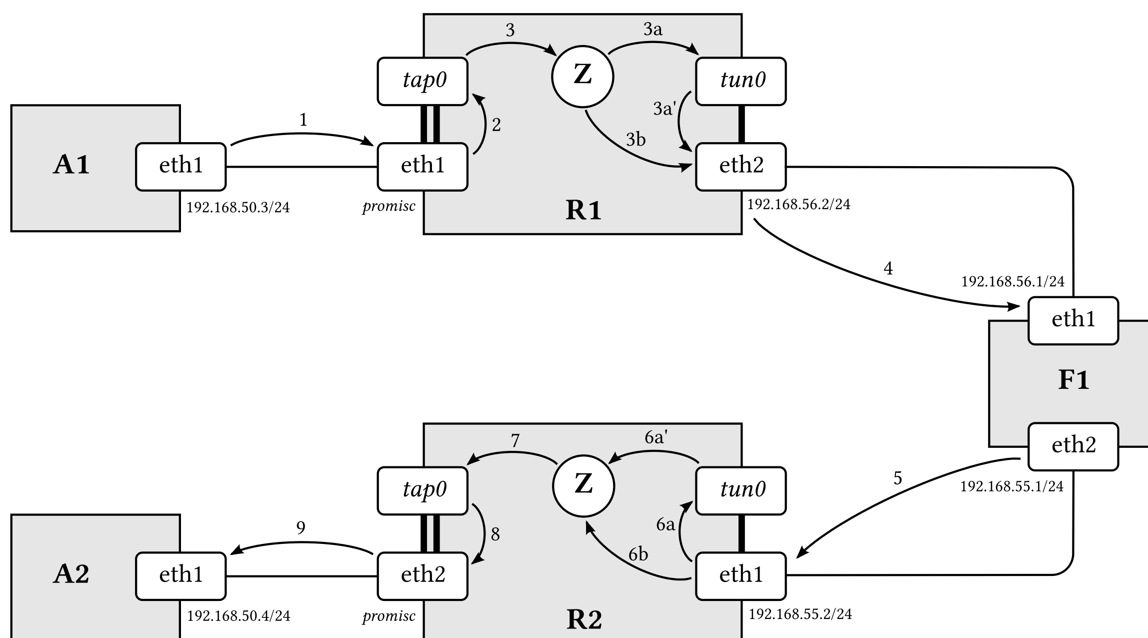
7. Zero serveris atjauno oriģinālo ethernet paketi un ievada to $tap0$ iekārtā.

8. $tap0$ ir savienots ar $eth2$ izmantojot *bridge*, līdz ar to, pakete nokļūst $eth2$ iekārtā.

9. Pakete tiek padota tālāk un nokļūst galamērķī – $A2$ datorā.

2.5. Zero servera uzstādīšana

Serveris ir programmēts ievērojot Python *de facto* pakotņu izplatīšanas metodi – izmantojot programmu `setuptools`. Zero serveris tiek izplatīts ar speciālu `tar.gz` failu, kurā ir



Att. 10: Zero servera darība pa soļiem.

programmas source, kā arī papildus informācija, kas vajadzīga to instalējot. Lai ieinstalētu Zero serveri, ir vajadzīgas Python `setuptools` vai `distribute` programmas. `setuptools` ir *de facto* standarts, `distribute` ir jauns risinājums, kas ir saderīgs ar `setuptools`. Sakarā ar to, ka šobrīd šīs izplatīšanas tehnoloģijas nomaina viena otru, dažkārt var rasties neskaidrības. Tātad, vajag vai nu vienu, vai otru.

Linux sistēmās šis pakotnes var atrast ar nosaukumiem

- `python-setuptools`
- `python-distribute`

Debian GNU/Linux sistēmās to var ieinstalēt ar vienu no komandām:

```
# apt-get install python-setuptools
# apt-get install python-distribute
```

setuptools un distribute piedāvā pakotņu instalēšanas komandas. Standarda komanda ir `easy_install`, bet varētu būt pieejama arī jaunāka `pip`. Jebkura no šīm komandām māk instalēt Zero serveri.

Operācija pieprasa sistēmas atļaujas, lai kopētu nepieciešamos failus vajadzīgajās vietās. Linux sistēmā parasti programmas tiek kopētas uz `/usr/local/bin`, Python pakotnes uz `/usr/local/lib/python2.7/site-packages` vai līdzīgi.

Ieinstalēt Zero var ar vienu no sekojošām komandām. Zero programmas versijas numurs varētu atšķirties.

```
# easy_install zero-0.1a16.tar.gz
# pip install zero-0.1a16.tar.gz
```

Zero serverim nāk līdzi skripts, kas izveido konfigurācijai nepieciešamās mapes, kā arī parauga uzstādījumu failu. Ieinstalējot Zero serveri, ir jābūt pieejamai komandai `zero-setup`. To vajadzētu palaist uzreiz pēc Zero ieinstalēšanas.

Īsumā instalāciju var veikt izpildot komandas:

```
# easy_install zero-0.1a16.tar.gz
# zero-setup
# vim ~/.zero/app.ini
```

Kā jau iepriekš minēts, lai darbinātu Zero serveri ir nepieciešama attiecīga tīkla konfigurācija. Tas, kā tīklu konfigurēt ir atkarīgs no sistēmas. Zero nāk līdzi skripts `zero-netconfig`, kas palīdz veikt tīklu konfigurāciju. Tas ir veidots ar domu par Linux sistēmu, kurā ir pieejamas programmas `ifconfig`, `brctl`, `route`, `sysctl`. Ja viss nepieciešamais ir pieejams, tad to

var izmantot, lai konfigurētu tīklu Zero darbībai. Skripts nolasa konfigurācijas failu, attiecīgi izveido TUN un TAP iekārtas, sagatavo *bridge*, nepieciešamos maršrutēšanas uzstādījumus, tas uzstāda arī IP *forwarding* ar `sysctl`. Attiecīgi skriptam nepieciešamas atļaujas.

Zero ir pievienots konfigurācijas fails, kas kalpo kā piemērs. Tajā ir redzami visi iespējamie konfigurācijas parametri. Daļa no šiem parametriem ir vajadzīgi tīkla konfigurācijā, daļa tikai Zero servera darbībai.

```
[zero]
pidfile = %(here)s/zero.pid
statsfile = %(here)s/stats.txt

host = 0.0.0.0
port = 8004

rhost = 192.168.56.2
rport = 8004

dest = 77
dest_in = 88

ttl_delta = 3

iftap = eth1
iftun = eth0

tundev = tun0
tapdev = tap0
brdev = br0

address = 192.168.55.2
netmask = 255.255.255.0
gateway =
```

pidfile servera procesa PID fails.

statsfile statistikas fails.

host Zero servera UDP hosts.

port Zero servera UDP ports.

rhost otra Zero servera adrese.

rport otra Zero servera UDP ports.

dest otra Zero servera /8 tīkls.

dest_in Zero servera /8 tīkls (tīkla uzstādīšanai).

ttl_delta sākotnējais TTL samazinājums.

iftap ar TAP iekārtu savienotā tīkla ierīce (tīkla uzstādīšanai).

iftun ar TUN savienotā tīkla ierīce (tīkla uzstādīšanai).

tundev TUN ierīces nosaukums.

tapdev TAP ierīces nosaukums.

brdev *bridge* ierīces nosaukums (tīkla uzstādīšanai).

address Zero adrese (tīkla uzstādīšanai).

netmask Zero adreses tīkla maska (tīkla uzstādīšanai).

gateway vārteja uz tuneļa otru galu, ja tāda ir (tīkla uzstādīšanai).

Zero serveri palaiž ar komandu `zero-server`. Ir pieejams neliels lietojuma apraksts padodot `-h`. Ir trīs izvēles parametri, servera konfigurācijas faila atrašanās vieta, servera *log* konfigurācijas fails (izmanto Python standarta *log* mehānismu un konfigurāciju), kā arī norāde, vai process ir jāpārvērš par *daemon*.

```
# zero-server -h
Usage: zero-server [options]

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -c APP_CONFIG, --app-config=APP_CONFIG
                        Application configuration [default:
                        /root/.zero/app.ini]
  -l LOG_CONFIG, --log-config=LOG_CONFIG
                        Logging configuration [default:
                        /root/.zero/log.ini]
  -d, --daemonize       Daemonize process [default: False]
```

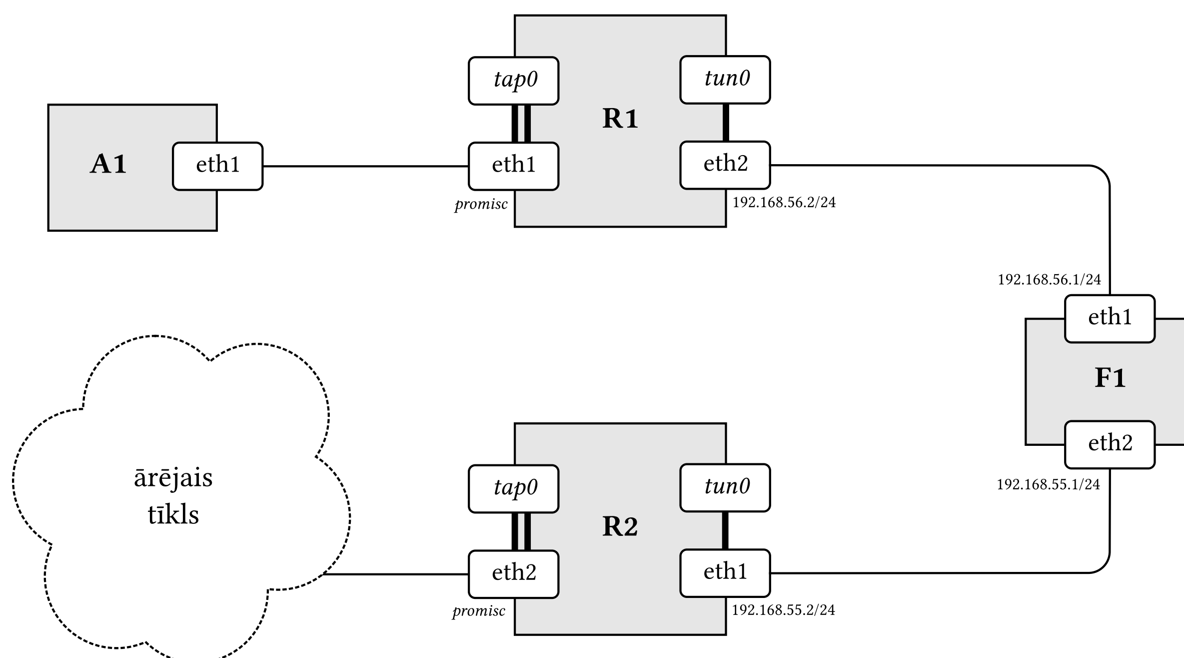
Lai ieinstalētu un uzstādītu Zero serveri, to var paveikt ar sekojošu secību. Pirmā komanda ieinstalē Zero serveri, otra izveido nepieciešamās mapes un iekopē parauga konfigurācijas failu. Pēc tam to atver ar teksta redaktoru un pielāgo vajadzībai. Pēc tam var izpildīt tīkla konfigurācijas skriptu. Visbeidzot, tiek palaists Zero serveris.

```
# easy_install zero-0.1a16.tar.gz
# zero-setup
# vim ~/.zero/app.ini
# zero-netconfig
# zero-server
```

3. STATISTIKA

3.1. Zero servera statistika

Eksperimentam uzbūvēta testa vide līdzīga nodaļā 2.3 aprakstītajam, izmantojot VirtualBox virtuālās mašīnas. Testa vide sastāv no 4 virtuālajām mašīnām. Viena no tām ir klients, kas piekļūst ārējam tīklam, tunelējot paketes cauri citam tīklam. Ir divi datori, kas kalpo kā tuneļa gali. Viens dators simulē tuneļa transporta tīkla sadalījumu divos tīklos. Attēlā 11 ir redzama šī eksperimenta konfigurācija.



Att. 11: IP galvene

Datori $R1$ un $R2$ ir tuneļa gali, kas nodrošina Zero protokolu. Katrā no tiem ir uzstādīts Zero serveris. $A1$ ir patvaļīgs dators bez papildus konfigurācijas. Eksperimentā tas tiek izmantots, lai piekļūtu globālajam tīmeklim. Lai nodarbotos ar ikdienas tīmekļa lietojumu – skatīties video, aplūkotu dažādas tīmekļa vietnes, lejuplādētu failus un tamlīdzīgi. Eksperiments ilgst apmēram astoņas minūtes, kurā Zero serveri datoros $R1$ un $R2$ savāc statistiku par tunelī ienākošām paketēm.

Zero serveri vāc datus par tunelī ienākošajām paketēm, kāds tām ir protokols, vai tās ir NICE IP, vai UGLY IP. Pa kādu Zero servera transporta kanālu tās tiek nosūtītas otrajam galam. Cik ir jaunas ienākošās ethernet galvenes, cik ir gadījumos izdevās paketi transportēt ar ZERO IP metodi un tamlīdzīgi.

Jāpiemin, kā šai komunikācijai datoram $A1$ ar globālo tīmekli ir nesimetriska daba. Tas

vairāk lejuplādē datus, augšupielāde faktiski nav. Līdz ar to, savāktie dati tuneļa galos $R1$ un $R2$ būs ļoti atšķirīgi. $R2$ ienāks daudz lielāks datu apjoms, salīdzinot to ar $R1$.

protokols	$R1$	$R2$
ARP	9	5
IP	61273	125472
IPV6	5	0

Tabula 1: Pakešu skaits tuneļa galos $R1$, $R2$ sadalīts pēc protokola.

statuss	R1	R2
NICE	61253	125472
UGLY	34	5

Tabula 2: Pakešu skaits tuneļa galos $R1$, $R2$ sadalīts pēc NICE un UGLY ethernet paketēm.

Aplūkojot tabulas 1 un 2, redzam, ka kopā $R1$ saņēma 61287, bet $R2$ 125477 ethernet paketes. No $R2$ ienākošajām IP paketes, visas ir NICE IP. $R1$ pusē arī ir ļoti liels pārsvars NICE IP paketēm. Tur no visām IP paketēm 99.97% ir NICE IP.

izeja	R1	R2
UDP	83	79
TUN	61204	125398

Tabula 3: Pakešu skaits, ko izdevās nosūtīt ar ZERO IP un ZERO UDP transporta metodi.

Izpētot tabulas 3 datus, redzam, ka lielāko daļu no paketēm izdevās transportēt optimizētā veidā bez pievienotajiem datiem. $R1$ galā tās ir 99.86%, bet $R2$ tuneļa galā tie ir 99.94%.

statuss	R1	R2
NICE-HAV	61204	125398
UGLY-PAK	34	5
NICE-NEW	3	2
NICE-MEM	46	72

Tabula 4: Pakešu sadalījums pēc to statusa.

Tabulā 4 ir attēlotai pakešu skaiti sadalīti pēc paketes statusa. Šeit apskatīti četri dažādi ienākošās ethernet paketes novērtējumi. NICE-HAV nozīmē, ka pakete ir NICE IP un tai atbilstošā ethernet galvene ir saglabāta tabulā. UGLY-PAK apzīmē UGLY ethernet paketes – vai nu tās nav IP, vai arī tās ir UGLY IP. NICE-NEW apzīmē gadījumus, kad ienākusi NICE IP pakete, bet tās ethernet galvene vēl nav saglabāta tabulā. NICE-MEM ir situācija, kad ienākusi NICE

IP pakete, bet tiek sūtīta pilna ethernet pakete ar ZERO UDP transportu, lai atgādinātu otram tuneļa galam ethernet tabulas ierakstu.

garums (baiti)	skaits
66	55351
78	3502
86	648
54	307
74	207
94	167
98	93
82	87
376	34
79	26

Tabula 5: $R1$ ienākošo ethernet pakešu biežāk sastopamie garumi.

garums (baiti)	skaits
1514	121185
1484	1800
66	431
74	155
316	116
1326	105
98	88
70	67
54	65
371	63

Tabula 6: $R2$ ienākošo ethernet pakešu biežāk sastopamie garumi.

Tabulā 5 attēlotas visbiežāk saņemot pakešu garumi $R1$. Kā jau iepriekš varēja prognozēt, šajā tuneļa galā neparādās garas paketes, jo tīkla lietojums ir vairāk vērsts uz to, ka lielākais apjoms ar datiem plūst virzienā no ārējā tīkla uz $A1$. $R1$ Zero serveris savāc statistiku par otra virziena datiem – no $A1$ uz ārējo tīklu.

Tabulā 6 attiecīgi ir attēlotas $R2$ gala biežāko pakešu garumi. Redzams, ka lielākā daļa no paketēm, kas nokļūst tunelī $R2$ pusē ir pilnas ethernet paketes. MTU ir standarta 1500, tas nozīmē, ka pilnas ethernet paketes garums ir 1500 baitu dati ar ethernet galveni – tie ir 1514 baiti.

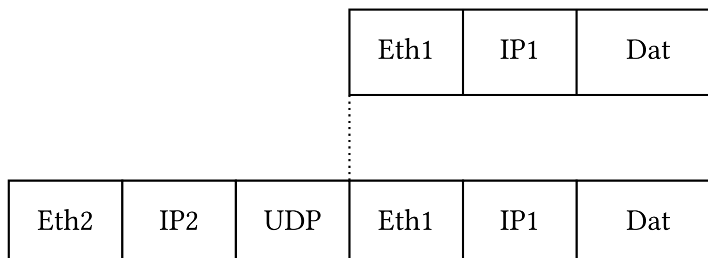
Maksimālā izmēra ethernet paketes $R2$ pusē ir 96,58% no visām ienākošajām paketēm. Pie tam tās visas ir NICE IP paketes, kuras var transportēt ar ZERO IP veidu, nepievienojot papildus datus.

Izpētot tabulu 4, varam redzēt, ka optimizētā veidā izdevās nosūtīt 99,94% no visām NICE IP paketēm. Tie nav 100%, jo ir jāsūta pilnas ethernet paketes ar ZERO UDP, lai paziņotu otrajam galam par ethernet galveņu tabulas skaitļiem.

3.2. Salīdzinājums ar OpenVPN tuneli

Lai vairāk akcentētu Zero protokola optimizācijas rezultātus, varam salīdzināt to ar parastu, neoptimizētu tunelēšanu. Kā, piemēram, to piedāvā OpenVPN. OpenVPN ir būvēts, lai nodrošinātu tunelēšanu nedrošā tīklā, paketes kriptējot, bet tas arī piedāvā izveidot parastu tuneli ar UDP protokola transportu.

Nodaļā 3.1 tika veikts eksperiments, lai ievāktu Zero servera darbības statistiku. Tos pašus datus varam izmantot, lai salīdzinātu Zero optimizēto tunelēšanu ar OpenVPN neoptimizēto. Tā kā aprakstītā Zero protokola realizācija – Zero serveris – nodrošina detalizētu statistikas vākšanu, tad varam precīzi analizēt situāciju. Zero serveris par katru paketi ieraksta informāciju, cik tai ir garums, vai tā ir NICE IP, ar kādu transportēšanas metodi tā tika nosūtīta otram tuneļa galam, utt.



Att. 12: Paketes tunelēšana ar UDP.

Tabulās 7 un 8 ir salīdzināta Zero server un OpenVPN tunelēšana. Tajā ir atainoti tunelī pārsūtīto pakešu skaits un kopējais izsūtīto pakešu datu apjoms baitos. Izmantoti Zero serveru savāktie dati no eksperimenta, kas aprakstīts nodaļā 3.1.

Datorā $R1$ Zero serverim nav liels ietaupījums uz pārsūtīto pakešu skaitu (tabula 7), jo šajā tuneļa pusē ienākošās paketes ir izmērā mazas, jo tāda ir eksperimentā veiktā tīkla noslodzes īpatnība. Pārsūtīto pakešu skaits atšķiras par 20. Zero protokolam nepieciešams par 20 paketēm mazāk. Tomēr $R1$ pusē Zero ietaupa vairāk uz pārsūtīto datu apjomu. Zero serveris izmanto tikai 63,27% no pakešu kopējā datu apjoma, kas vajadzīgs OpenVPN.

Tuneļa otrā galā ir citādāka situācija (tabula 8). Šajā virzienā tiek sūtītas lielas ethernet paketes, kā arī daudz skaitā. Redzam milzīgu ietaupījumu pakešu skaita ziņā. Zero proto-

kols izmanto 50,5% no OpenVPN tuneļa pakešu skaita. Tik liela atšķirība, jo garās paketes ir jāsadala mazākās daļās. Šeit gan novērojam mazāku atšķirību pārsūtīto datu apjomā, bet tas ir loģiski, jo protokolu galveņu dati šeit nedod lielu devumu, salīdzinot ar pašu protokolu transportēto informāciju.

	Zero	OpenVPN
pakešu skaits	61289	61309
apjoms baitos	4430144	7001420

Tabula 7: $R1$ tunelī izsūtīto pakešu statistikas salīdzinājums Zero ar OpenVPN.

	Zero	OpenVPN
pakešu skaits	125491	248473
apjoms baitos	187155054	197586861

Tabula 8: $R2$ tunelī izsūtīto pakešu statistikas salīdzinājums Zero ar OpenVPN.

SECINĀJUMI

Darbā tika veiksmīgi izstrādāts Zero tunelēšanas protokols ar tā realizāciju GNU/Linux operētājsistēmā izmantojot valodu Python. Tika uzkonstruētas virtuālas vides, kurā šī Zero servera realizācija tika pārbaudīta dažādos scenārijos, ievākti statistikas dati par tā darbību.

Secināts, ka protokols spēj nodrošināt iecerēto optimizāciju. Eksperimentos gūta pārliecība, ka tas pārliecinoši lielāko daļu pakešu var apstrādāt optimizētā veidā, nepievienojot papildus transportēšanas datus paketei.

Iesākto paredzēts turpināt. Nepieciešams vairāk izpētīt to darbību pie dažāda veida tīkla plūsmām. Cik daudz tas var optimizēt paketes ar VLAN, MPLS un dažādiem to kombināciju marķējumiem. Kad protokola detaļas būs pilnībā skaidras, tā realizāciju jāpārraksta valodā C, lai iegūtu maksimālu ātrdarbību.

Par izstrādāto risinājumu interesējas EADS Astrium kompānija Vizada Networks, kas arī izvērtē šo risinājumu ieviešanu praksē.

LITERATŪRAS SARAKSTS

[1] W. Richard Stevens, Bill Fenner, Andrew M. Rudoff, *Unix Network Programming, Volume 1: The Sockets Networking API (3rd Edition)*, Addison Wesley, 2003.

[2] OSI model, http://en.wikipedia.org/wiki/OSI_model, [2012-05-19]

[3] User space, http://en.wikipedia.org/wiki/User_space, [2012-05-19]

[4] TUN/TAP, http://en.wikipedia.org/wiki/TUN/TAP, [2012-05-19]

[5] Python Advocacy HOWTO, http://docs.python.org/howto/advocacy.html#prototyping, [2012-05-19]

[6] OpenVPN, http://en.wikipedia.org/wiki/OpenVPN, [2012-05-19]

[7] MTU, http://en.wikipedia.org/wiki/Maximum_transmission_unit, [2012-05-19]

[8] Ethernet II, http://en.wikipedia.org/wiki/Ethernet_frame#Ethernet_II, [2012-05-19]

[9] VLAN, http://en.wikipedia.org/wiki/Vlan, [2012-05-19]

[10] MPLS, http://en.wikipedia.org/wiki/Multiprotocol_Label_Switching, [2012-05-19]

[11] Evil bit, http://en.wikipedia.org/wiki/Evil_bit, [2012-05-19]

[12] packet(7), *Linux Programmer's Manual*, release 3.35, 2008-08-08.

[13] The Linux Kernel Module Programming Guide, http://tldp.org/LDP/lkmpg/2.6/html/lkmpg.html, [2012-05-19]

A. BŪTISKAIS PROGRAMMAS KODS

Kods 1: Zero servera handle funkcijas (zero/handles.py)

```
1  import os
2  import struct
3  import logging
4
5  from zero.formats import chunk_format, ip_packet_str
6  from zero.const import ETH_PROTOS
7  from zero.chtable import FullChunkTable, SimpleChunkTable
8  from zero.stats import Stats
9  from zero.packet import nice_ip_find
10 from zero.checksum import checksum
11
12 log = logging.getLogger(__name__)
13
14 chunk_table_this = FullChunkTable()
15 chunk_table_other = SimpleChunkTable()
16
17 stats = Stats()
18
19 # Maximum packet size to read.  Let it be more than jumbo frame.
20 PACKET_MAX = 10000
21
22 def handle_tap_read(server, tap_fd, tun_fd, dest, usock_r, rhost, rport):
23     log.debug(">>> TAP read")
24     packet = os.read(tap_fd, PACKET_MAX)
25     log.debug("%d bytes", len(packet))
26
27     stats.set_input_type("TAP")
28     stats.set_length(len(packet))
29
30     proto = struct.unpack("!H", packet[12:14])[0]
31     stats.set_proto(ETH_PROTOS.get(proto, "0x%02x" % proto))
32
33     tun_send_ttl_if_time(tun_fd, dest)
34
35     # Check if NICE and get header chunk.
36     is_nice = False
37     nice_list = nice_ip_find(packet)
38     l = len(nice_list)
39     ehdr = None
40
41     ttl_delta = server.ttl_delta
42
43     if l > 0:
44         is_nice = True
45         ehdr = packet[:nice_list[0]]
```

```
46
47          # Check if TTL not too small.
48          iph = packet[len(ehdr):]
49          ttl = struct.unpack("B", iph[8])[0]
50          if ttl <= ttl_delta:
51              log.debug("Packet not Nice because of TTL (%d)", ttl)
52              is_nice = False
53
54      # Send encapsulated via UDP.
55      if not is_nice:
56          log.debug("Packet is UGLY")
57          stats.set_packet_type("UGLY-PAK")
58          stats.set_nice_status("UGLY")
59          _send_ugly_chunk(usock_r, rhost, rport, packet)
60          return
61
62      # Packet is Nice.
63      log.debug("Packet is NICE")
64      stats.set_nice_status("NICE")
65      assert ehdr is not None
66
67      # Clean rare headers.  TODO: where to put this?
68      chunk_table_this.clean_rare_if_needed()
69
70      # 1. We have header.
71      #    a. Header memorized.
72      #    b. Header not memorized.
73      # 2. We don't have header.
74
75      # We have header in the table.  But we send number only after a while.
76      if chunk_table_this.have_chunk(ehdr):
77          num = chunk_table_this.get_num(ehdr)
78          log.debug("Have chunk %d", num)
79          log.debug(chunk_format(ehdr))
80
81          chunk_table_this.inc(num)
82          log.debug("Chunk count: %d", chunk_table_this.get_cnt(num))
83
84          # We think header have been send to other side.
85          if chunk_table_this.memorized(num):
86              log.debug("Memorized")
87              log.debug("Send modified")
88
89              # Send modified IP packet.
90              tun_send(tun_fd, dest, num, packet[len(ehdr):])
91
92          # If is possible that other side have no header yet.
93          else:
```

```
94                log.debug("Not memorized")
95                log.debug("Send full packet")
96
97                stats.set_packet_type("NICE-MEM")
98                _send_new_chunk(usock_r, rhost, rport, num, packet)
99
100       # Save header in table and notify other end, send full packet.
101       else:
102           log.debug("Save new chunk")
103           log.debug(chunk_format(ehdr))
104
105           if chunk_table_this.is_full():
106               log.error("Chunk table full")
107               stats.set_packet_type("NICE-FUL")
108               _send_ugly_chunk(usock_r, rhost, rport, packet)
109               return
110
111           num = chunk_table_this.save(ehdr)
112           log.debug("Saved with number %d", num)
113
114           # Send number and full packet.
115           log.debug("Send full packet")
116           stats.set_packet_type("NICE-NEW")
117           _send_new_chunk(usock_r, rhost, rport, num, packet)
118
119   def _send_ugly_chunk(usock_r, rhost, rport, packet):
120       hdr = struct.pack("B", 1)
121       data = hdr + packet
122
123       stats.set_output_type("UDP")
124       stats.set_output_length(len(data))
125
126       # XXX: Add stats.
127       stats.add()
128
129       log.debug("*** UDP write (%d bytes)", len(data))
130       usock_r.sendto(data, (rhost, rport))
131
132   def _send_new_chunk(usock_r, rhost, rport, num, packet):
133       hdr = struct.pack("BB", 0, num)
134       data = hdr + packet
135
136       stats.set_output_type("UDP")
137       stats.set_output_length(len(data))
138
139       # XXX: Add stats.
140       stats.add()
141
```

```
142     chunk_table_this.set_sync(num) # Update sync time.
143
144     log.debug("*** UDP write (%d bytes)", len(data))
145     usock_r.sendto(data, (rhost, rport))
146
147 def handle_udp_read(usock, tap_fd):
148     """Receive Ugly packet or IP packet with new (unsaved) ethernet header.
149     """
150
151     log.debug(">>> UDP read")
152     packet = usock.recv(PACKET_MAX)
153     log.debug("%d bytes", len(packet))
154
155     # Our header info.
156     stat = struct.unpack("B", packet[:1])[0]
157
158     # 1. Packet is Nice.  We get number and save ethernet header to table.
159     # Write full packet.
160     if stat == 0:
161         num = struct.unpack("B", packet[1:2])[0]
162
163         log.debug("NICE packet")
164         log.debug("New chunk %d", num)
165
166         pack = packet[2:]
167
168         nice_list = nice_ip_find(pack)
169         assert len(nice_list) > 0
170         ehdr = pack[:nice_list[0]]
171
172         log.debug(chunk_format(ehdr))
173
174         # Regardless what we have, we save this.  It can either be repeated
175         # send or record overwrite with new value or new record.
176         chunk_table_other.save(num, ehdr)
177
178         data = pack
179         log.debug("*** TAP write (%d bytes)", len(data))
180         os.write(tap_fd, data)
181
182     # 2. Packet is Ugly.  Just write full packet.
183     elif stat == 1:
184         log.debug("UGLY packet")
185
186         data = packet[1:]
187         log.debug("*** TAP write (%d bytes)", len(data))
188         os.write(tap_fd, data)
189
```

```
190     else:
191         raise Exception("Unknown packet status: %s" % stat)
192
193 def _handle_ttl(server, packet):
194     ttl = struct.unpack("B", packet[8])[0]
195     delta = 255 - ttl
196     log.debug("Received TTL (TTL delta = %d)", delta)
197     server.ttl_delta = delta
198
199 def handle_tun_read(server, tun_fd, tap_fd):
200     """Receive modified IP packet with destination bits and ethernet header
201     number hidden in fragmentation field.  We receive from other end.
202     """
203
204     log.debug(">>> TUN read")
205     packet = os.read(tun_fd, PACKET_MAX)
206     log.debug("%d bytes", len(packet))
207
208     packet_str = ip_packet_str(packet)
209     log.debug("\n" + packet_str)
210
211     c = struct.unpack("B", packet[0])[0]
212     version = c >> 4
213     ihl = c & 0b00001111
214
215     # Get IP protocol number.
216     proto = struct.unpack("B", packet[9])[0]
217     if proto == 253: # TTL announcement mark (experimental protocol Nr).
218         _handle_ttl(server, packet)
219         return
220
221     # Assert IPv4 without options.
222     assert version == 4
223     assert ihl == 5
224
225     # Assert our modified.
226     c = struct.unpack("B", packet[6])[0]
227     flags = c >> 5
228
229     f_re = (flags >> 2) & 1
230     #f_df = (flags >> 1) & 1
231     f_mf = (flags >> 0) & 1
232
233     # Assert our modified IP packet.
234     assert f_re == 1
235     log.debug("Evil bit has been set")
236     assert f_mf == 0
237
```

```
238     # Here lays:
239     # 1) ethernet header number,
240     # 2) real destination first 8 bits.
241     c = struct.unpack("!H", packet[6:8])[0]
242     frag_offset = c & 0b0001111111111111
243
244     # First 5 bits -- ethernet header number.
245     # Last 8 bits -- first bits of real destination.
246     num = frag_offset >> 8
247     dest = frag_offset & 0b0000011111111
248
249     log.debug("Chunk num: %d", num)
250     log.debug("Dest byte: %d", dest)
251
252     # Get back original packet.
253     try:
254         ehdr = chunk_table_other.get_chunk(num)
255     except KeyError:
256         log.error("No chunk with number %d", num)
257         return
258
259     # Set all three flags and fragment offset to null.
260     b1 = 0b010 << 5
261     packet = packet[:6] + struct.pack("BB", b1, 0) + packet[8:]
262
263     # Set real IP destination.  Change back first byte of destination.
264     packet = packet[:16] + struct.pack("B", dest) + packet[17:]
265
266     # Compensate TTL.
267     ttl = struct.unpack("B", packet[8])[0]
268     ttl_delta = server.ttl_delta
269     packet = packet[:8] + struct.pack("B", ttl + ttl_delta) + packet[9:]
270
271     packet = fix_ip_checksum(packet)
272
273     packet_str = ip_packet_str(packet)
274     log.debug("\n" + packet_str)
275
276     # Write to TAP.
277     data = ehdr + packet
278     log.debug("*** TAP write (%d bytes)", len(data))
279     os.write(tap_fd, data)
280
281 _ip_packet = [ 0x45, 0x0, 0x0, 0x14, 0x0, 0x0, 0x0, 0x0, 0xff, 0xfd, 0xba,
282 0xed, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0 ]
283 _ip_packet = "".join([chr(x) for x in _ip_packet])
284
285 def tun_send_ttl(tun_fd, dest):
```

```
286     """Send specially formed IP packet to inform other side about TTL value to
287     compensate.  We send TTL 255 in the other end we substract actual TTL from
288     255 and get delta.
289     """
290 
291     log.debug("Send TTL")
292 
293     packet = _ip_packet
294 
295     # Set protocol.
296     proto = 253
297     packet = packet[:9] + struct.pack("B", proto) + packet[10:]
298 
299     # Set TTL.
300     ttl = 255
301     packet = packet[:8] + struct.pack("B", ttl) + packet[9:]
302 
303     # Set valid IP source and destination.
304     d = [dest, 0, 0, 2]
305     packet = packet[:12] + struct.pack("BBBB", *d) + packet[16:]
306     d = [dest, 0, 0, 1]
307     packet = packet[:16] + struct.pack("BBBB", *d)
308 
309     # Fix checksum.
310     packet = fix_ip_checksum(packet)
311 
312     log.debug("/// TUN write (%d bytes)", len(packet))
313     os.write(tun_fd, packet)
314 
315 import time
316 
317 last_ttl_sent = 0
318 
319 def tun_send_ttl_if_time(tun_fd, dest):
320     global last_ttl_sent
321     t = int(time.time())
322     if t - last_ttl_sent > 10:
323         tun_send_ttl(tun_fd, dest)
324         last_ttl_sent = t
325 
326 def tun_send(tun_fd, dest, num, packet):
327     """Send modified Nice IP packet via TUN with ehernet header number num.
328     """
329 
330     assert 0 <= num < 32
331     assert 0 <= dest < 255
332 
333     log.debug("Send modified packet via TUN")
```

```
334
335     packet_str = ip_packet_str(packet)
336     log.debug(”\n” + packet_str)
337
338     log.debug(”Chunk num: %d”, num)
339
340     # Get packet flags.
341     b1, b2 = struct.unpack(”BB”, packet[6:8])
342     flags = b1 >> 5
343
344     f_re = (flags >> 2) & 1
345     f_df = (flags >> 1) & 1
346     f_mf = (flags >> 0) & 1
347
348     assert f_re == 0
349     assert f_mf == 0
350
351     # Set Evil (reserved) bit to mark our packet.
352     flags = (1 << 2 | f_df << 1 | f_mf)
353
354     # Set first 5 bits of fragment offset to num.
355     b1 = flags << 5 | num # First byte.
356
357     # Copy first 8 bits of destination field.
358     d = packet[16:20]
359
360     b2 = d[0] # Second byte.
361     log.debug(”Dest byte: %d”, ord(b2))
362
363     # Sef fragment offset field.
364     packet = packet[:6] + struct.pack(”BB”, b1, ord(b2)) + packet[8:]
365
366     # Change first 8 bits of destination.
367     packet = packet[:16] + struct.pack(”B”, dest) + packet[17:]
368
369     packet = fix_ip_checksum(packet)
370
371     packet_str = ip_packet_str(packet)
372     log.debug(”\n” + packet_str)
373
374     stats.set_packet_type(”NICE-HAV”)
375     stats.set_output_type(”TUN”)
376     stats.set_output_length(len(packet))
377
378     # XXX: Add stats.
379     stats.add()
380
381     # Write to TUN.
```

```
382     log.debug("*** TUN write (%d bytes)", len(packet))
383     os.write(tun_fd, packet)
384
385 def fix_ip_checksum(packet):
386     # Set checksum to null.
387     x = struct.pack("!H", 0)
388     packet = packet[:10] + x + packet[12:]
389
390     # Calculate checksum.
391     iph = packet[:20]
392     c = checksum(iph)
393     packet = packet[:10] + struct.pack("H", c) + packet[12:]
394     return packet
```

Kods 2: Pieslēgšanās TUN/TAP iekārtām (zero/tuntap.py)

```
 1 import os
 2 import struct
 3 import logging
 4 from fcntl import ioctl
 5
 6 log = logging.getLogger(__name__)
 7
 8 # TUNSETIFF ifr flags
 9 IFF_TUN = 0x0001
10 IFF_TAP = 0x0002
11 IFF_NO_PI = 0x1000
12 IFF_ONE_QUEUE = 0x2000
13 IFF_VNET_HDR = 0x4000
14 IFF_TUN_EXCL = 0x8000
15
16 TUNSETIFF   = 0x400454ca
17
18 def tun_alloc(dev, flags):
19     log.debug("TUN alloc")
20     fd = os.open("/dev/net/tun", os.O_RDWR)
21
22     # Set device name and flags.
23     data = struct.pack("16sH", dev, flags)
24
25     # Attach to device.
26     ifs = ioctl(fd, TUNSETIFF, data)
27
28     device_name = ifs[:16].strip("\x00")
29     log.debug("Using interface %s", device_name)
30
31     return fd
```

Kods 3: Zero servera startēšanās kods (zero/scripts.py)

```
def server():
    opt_parser = OptionParser(usage="%prog [options]",
        version=__version__,
        )

    opt_parser.add_option("-c", "--app-config", default=app_config_path,
        help="Application configuration [default: %default]")
    opt_parser.add_option("-l", "--log-config", default=log_config_path,
        help="Logging configuration [default: %default]")
    opt_parser.add_option("-d", "--daemonize", action="store_true",
        default=False, help="Daemonize process [default: %default]")

    options, args = opt_parser.parse_args()
    if len(args) != 0:
        opt_parser.print_usage()
        exit(2)

    app_conf_path = options.app_config
    log_conf_path = options.log_config
    daemonize = options.daemonize

    #logging_config(log_conf_path)
    settings = get_app_settings(app_conf_path)

    tap_dev = settings["tapdev"]
    tun_dev = settings["tundev"]
    host = settings["host"]
    port = int(settings["port"])
    rhost = settings["rhost"]
    rport = int(settings["rport"])
    dest = int(settings["dest"]) # IP destination byte.
    pidfile_name = settings["pidfile"]
    statsfile_name = settings["statsfile"]
    ttl_delta = int(settings["ttl_delta"])

    if daemonize:
        ctx = DaemonContext(
            pidfile=PidfileContext(pidfile_name),
            #stderr=sys.stderr,
            #stdout=sys.stdout,
            )
    else:
        ctx = NoDaemonContext(
            pidfile=PidfileContext(pidfile_name),
            )

    with ctx:
```

```
48          logging_config(log_conf_path)
49          log.info("Process running with pid = %d", os.getpid())
50          server = Server(tap_dev, tun_dev, dest, host, port, rhost, rport,
51              ttl_delta)
52          try:
53              server.serve_forever()
54          except (Exception, KeyboardInterrupt, SystemExit), e:
55              log.exception(e)
56              log.info("Saving stats")
57              with open(statsfile_name, "w") as fil:
58                  stats.write(fil)
59              log.info("Ending")
```

Kods 4: Zero servera kods (zero/server.py)

```
1  import logging
2  import socket
3  import select
4
5  from zero.tuntap import tun_alloc
6  from zero.tuntap import IFF_TAP, IFF_TUN, IFF_NO_PI
7  from zero.handles import handle_tap_read, handle_udp_read, handle_tun_read
8
9  log = logging.getLogger(__name__)
10
11 class Server(object):
12
13     def __init__(self, tap_dev, tun_dev, dest, host, port, rhost, rport,
14             ttl_delta):
15
16         self.rhost = rhost
17         self.rport = rport
18         self.dest = dest # Mod. IP send destination byte.
19
20         # Amount to compensate incomming packet TTL value.  We update this on
21         # special TTL 255 package receive via TUN.
22         self.ttl_delta = ttl_delta
23
24         # Create remote UDP socket.
25         log.debug("Create remote UDP socket")
26         self.usock_r = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
27
28         # Attach TAP device.
29         log.debug("Attach to TAP device '%s'" % tap_dev)
30         self.tap_fd = tun_alloc(tap_dev, IFF_TAP | IFF_NO_PI)
31
32         # Attach TUN device.
33         log.debug("Attach to TUN device '%s'" % tun_dev)
34         self.tun_fd = tun_alloc(tun_dev, IFF_TUN | IFF_NO_PI)
```

```
35
36          # Create our UDP socket.
37          log.debug("Create listening UDP socket")
38          self.usock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
39          self.usock.bind((host, port))
40
41      def serve_forever(self):
42          # Reading incommig packets.
43          log.info("Server started")
44          log.debug("Waiting for packets")
45
46          tap_fd = self.tap_fd
47          tun_fd = self.tun_fd
48
49          usock_r = self.usock_r
50          usock = self.usock
51
52          rhost = self.rhost
53          rport = self.rport
54
55          dest = self.dest
56
57          while True:
58              rd, wd, xd = select.select([tap_fd, tun_fd, usock], [], [])
59              for r in rd:
60                  if r == tap_fd:
61                      handle_tap_read(self, r, tun_fd, dest, usock_r, rhost,
62                          rport)
63                  elif r == usock:
64                      handle_udp_read(r, tap_fd)
65                  elif r == tun_fd:
66                      handle_tun_read(self, r, tap_fd)
67                  else:
68                      raise Exception("Unknown select descriptor")
```

Kods 5: Kontrolsummu rēķināšana (zero/checksum.py)

```
 1  import struct
 2
 3  def carry_around_add(a, b):
 4      c = a + b
 5      return (c & 0xffff) + (c >> 16)
 6
 7  def checksum(msg):
 8      s = 0
 9      for i in range(0, len(msg), 2):
10          w = ord(msg[i]) + (ord(msg[i+1]) << 8)
11          s = carry_around_add(s, w)
12      return ~s & 0xffff
```

```
13
14 def ip_checksum(iph):
15     """Calculate IP checksum.
16     """
17
18     x = struct.pack("!H", 0)
19     iph = iph[:10] + x + iph[12:]
20     csum = checksum(iph)
21
22     # Swap byte order.
23     h = csum >> 8
24     t = csum & 255
25     c = t << 8 | h
26
27     return c
```

Kods 6: Labo IP un eterhent pakešu meklēšana (zero/packet.py)

```
 1 import re
 2 import struct
 3 import logging
 4
 5 from zero.checksum import ip_checksum
 6
 7 full_re = re.compile(b"\x45.....(\x00|\x40)\x00", re.DOTALL)
 8
 9 log = logging.getLogger(__name__)
10
11 def nice_ip_marks(packet):
12     """Find NICE IP packets, return list of matched indexes.
13     """
14
15     m = full_re.finditer(packet)
16     return [x.start() for x in m]
17
18 def nice_ip_find(packet):
19     """Return list of NICE IP indication positions in packet.
20     """
21
22     nl = nice_ip_marks(packet)
23     nl = nl[:1] # Check only first indication.
24     return real_nice_ip_marks(packet, nl)
25
26 def _nice_check(packet, i):
27     ehdr = packet[:i]
28     rest = packet[i:]
29     iph = rest[:20]
30
31     if len(iph) < 20:
```

```
32            return False
33
34        # We can check here it IP total length is sane.  It should be less or equal
35        # to 1500 bytes and more than 20 bytes (minimal).
36
37        totlen = struct.unpack("!H", iph[2:4])[0]
38        l = len(packet) - len(ehdr)
39        diff = totlen - l
40
41        totlen_ok = totlen <= 1500 and totlen >= 20
42        diff_ok = diff == 0
43
44        # We could also check if IP checksum is correct.  If we do this can we
45        # forget about length checking?
46        cksum = struct.unpack("!H", iph[10:12])[0]
47        c = ip_checksum(iph)
48        checksum_ok = cksum == c
49
50        # Ethernet trailer should not be carried along when passing to bridged
51        # device?
52        if totlen_ok and checksum_ok and not diff_ok:
53            log.warning("Ugly trailer")
54
55        return totlen_ok and diff_ok and checksum_ok
56
57    def real_nice_ip_marks(packet, index_list):
58        """Additional checks to filter NICE IP indication list.
59        """
60
61        nice_list = list()
62        for i in index_list:
63            if _nice_check(packet, i):
64                nice_list.append(i)
65        return nice_list
```