NTNU
Det skapende universitet

Skjult IP-kommunikasjon basert på Domain Name System (DNS)

Lasse Karstensen

Master i kommunikasjonsteknologi
Oppgaven levert: Juli 2010
Hovedveileder: Stig Frode Mjølsnes, ITEM
Biveileder(e): Svein Yngar Willassen, ITEM

Norges teknisk-naturvitenskapelige universitet
Institutt for telematikk

Oppgavetekst

Programvare som iodine og TUNS implementerer metoder for å få en fullverdig IP-forbindelse gjennom oppslagstjenesten Domain Name System (DNS). Ettersom DNS er grunnleggende for en fungerende Internettforbindelse vil dette ofte kunne brukes som en skjult kommunikasjonskanal (covert channel) inn og ut av adgangsbegrensede nettverk.

Studenten skal sette seg inn i teknikkene relatert til IP over DNS med fokus på informasjonssikkerhet. Metoder for å detektere bruk av slike tunneler skal undersøkes og forsøkes videreutviklet.

Er dette en trussel for informasjonssikkerheten i bedrifters nettverk, eller bare en mindre irritasjon for operatører av trådløse nettverk på hoteller/flyplasser? Hva slags metoder kan bedrifter benytte for å beskytte seg mot slike tunneller?

Oppgaven gitt: 30. april 2010
Hovedveileder: Stig Frode Mjølsnes, ITEM

Forord

Dette er rapporten fra den avsluttende masteroppgaven i sivilingeniørstudiet i Kommunikasjonsteknologi på Norges Teknisk-naturvitenskapelige Universitet (NTNU).

Oppgaven min er å undersøke IP-tunnellering over DNS med hovedfokus på hva slags innvirkning det kan ha for informasjonssikkerhet. Denne teknikken er ikke så godt kjent i sikkerhetsmiljøene, og derfor heller ikke så godt undersøkt. Underveis har hovedfokuset blitt på deteksjon av dette gjennom en navnetjener. Dette har ingen gjort før, og å bruke tid på noe helt nytt har vært virkelig givende.

Dette har vært en spennende tid. Det å produsere et akademisk arbeid har vært en flott utfordring. Å formulere seg kortfattet og presist samtidig som man får kommunisert essensen i teksten har jeg lært en del om. Jeg håper jeg har fått dette til, og at leseren får utbytte av mitt arbeid.

Øystein Viggen hos NTNU IT-avdelingen skal ha takk for å hjelpe meg med pakkedumper fra en av NTNU sine DNS-servere.

Jeg retter en takk til min veileder Svein Yngvar Willassen og professor Stig Frode Mjølsnes for hjelp underveis.

Lasse Karstensen

Trondheim, 9. juli 2010

Innhold

Figurer

Tabeller

Sammendrag

I denne rapporten undersøkes det om det er mulig å detektere IP-over-DNS -trafikk gjennom en navnetjener automatisk. Bakgrunn for oppgavevalget var et utsagn av Nussbaum m.fl. som sa at eneste mulighet en systemadministrator hadde for å begrense IP-over-DNS generelt var båndbreddebegrensninger av klienter. Dette er undersøkt i laboratoriet, og alle kjente måter for å gjøre IP-over-DNS er prøvd ut eksperimentelt.

Rapporten gir to hovedbidrag: 1) det argumenteres for at det er mulig å detektere IP-over-DNS -trafikk automatisk og uten særlig mange falske alarmer, samt 2) gruppering av båndbreddeforbruk per mottakerdomene ser ut til å være den beste teknikken for å gjøre dette.

Arbeidet er gjort eksperimentelt og iterativt, og kildekode som implementerer samtlige utprøvde teknikker er vedlagt.

Kapittel 1

Introduksjon

1.1 Oppgavetekst

Programvare som iodine og TUNS implementerer metoder for å få en fullverdig IP-forbindelse gjennom oppslagstjenesten Domain Name System (DNS). Ettersom DNS er grunnleggende for en fungerende Internettforbindelse vil dette ofte kunne brukes som en skjult kommunikasjonskanal (covert channel) inn og ut av adgangsbegrensede nettverk.

Studenten skal sette seg inn i teknikkene relatert til IP over DNS med fokus på informasjonssikkerhet. Metoder for å detektere bruk av slike tunneller skal undersøkes og forsøkes videreutviklet.

Er dette en trussel for informasjonssikkerheten i bedrifters nettverk, eller bare en mindre irritasjon for operatører av trådløse nettverk på hoteller/flyplasser? Hva slags metoder kan bedrifter benytte for å beskytte seg mot slike tunneller?

1.2 Tolkning av oppgavetekst

Stallings [Sta05] definerer en skjult kommunikasjonskanal ("covert channel") som:

> A covert channel is a means of communication in a fashion unintended by the designers of the communications facility. Typically, the channel is used to transfer information in a way that violates a security policy.

Dette kan oversettes til at en skjult kanal er en kommunikasjonskanal som benytter mediet på et vis som ikke var opprinnelig planlagt til å kommunisere.

Motivasjon for bruk av IP over DNS
Hvorfor ønsker man å kjøre IP over DNS? I bunn og grunn ligger det et ønske om å få kommunisert med en annen part, som oftest ute på Internett. Den vanligste argumentasjonen er å unngå å måtte betale for Internett-tilgang på flyplasser og kafeer. Dette oppfattes av mange som ganske harmløst.

Samtidig har man ønsket om å snakke med en maskin som ikke bare er bak en brannvegg, men som er bak en brannvegg som ikke engang tillater utgående trafikk. Det er her IP-over-DNS, på godt og vondt kan bidra til å "grave hull" i brannveggen.

Av disse to situasjonene er det nok den første som er mest i bruk. Den andre vil vi sannsynligvis ikke høre om, om den nå er tilfelle ved datainnbrudd i bedrifter.

Om så hovedmotivasjonen bak bruk av IP-over-DNS er å spare seg en femtilapp eller å unngå å måtte bruke en veldig nedsperret Internett-forbindelse på flyplassen, bør dette være en indikator på hvor mye innsats det er verdt for en "angriper" å legge i å få det til å virke.

Kan det være at gjennom å gjøre det bare litt surere å få til IP-over-DNS, så blir det plutselig ikke-attraktivt? Hvor går i så fall denne grensen? Går det an å nå den uten å påvirke tjenestenivået til de andre betalende brukerene nevneverdig?

På den andre siden, er det mulig for en angriper å bruke DNS-basert tunnellering for å få en forbindelse som både er passelig vanskelig å sperre for, og som gir tilfredsstillende ytelse til interaktiv bruk?

Disse spørsmålene finner jeg interessante og skal forsøke å belyse i denne oppgaven.

1.3 Forutsetninger om omliggende miljø

For å konkretisere oppgaven er følgende bruksscenarier tenkt.

- Klassisk "hotspot", et trådløst nettverk der man må kjøpe seg tilgang for en tid.
- Bedriftsnettverk av ulik størrelse. I Norge er en vanlig bedrift ganske liten, og ressursene til proaktivt IT-sikkerhetsarbeid er derfor små. Vil IP-over-DNS kunne brukes her? Om det ikke er gjort så mye sikkerhetsarbeid, er det kanskje andre og mer effektive teknikker som kan brukes i stedet?

1.4 Alternativer til IP over DNS

De finnes mange andre metoder for å danne skjulte kanaler ut av begrensede nettverk. Eksempler på dette er å bruke ICMP-pakker til å tunnellere IP-pakker inni, slik beskrevet i [ASON03]. Videre har de fleste nettverk nå tilgang, enten direkte eller via en proxy, på World Wide Web (WWW) over HTTP/HTTPS. Om HTTPS er tillatt har man jo en ferdig kryptert kanal ut, og kommunikasjonsønsket til angriperen er løst. Om bare HTTP tillates, og kanskje gjennom en intelligent og innholdsfiltrerende proxy, finnes det metoder for å bruke ulike headerfelt i HTTP som bærekanal for den skjulte kanalen.

Alt i alt er IP over DNS bare ett av de mange problemene en sikkerhetspolicy i en bedrift må ta høyde for, men den er også en av de mer uhåndterlige ettersom teknikken er ukjent for de fleste og kjente teknikker for å sperre for det er få.

1.5 Hva skal undersøkes

Hypotesen som skal undersøkes er:

> Er det med automatiske teknikker mulig å entydig, eller med stor grad av sikkerhet, bestemme om en DNS-klient benytter IP-over-DNS?

En videre praktisk begrensning er her at dette må bestemmes ut fra nettverkstrafikken til den lokale (rekurserende) navnetjeneren. I dette tilfellet kan man enten speile all trafikk til DNS-serveren, eller rett og slett installere programvare på selve DNS-serveren som utfører oppgaven.

Merk at hvordan klienter som bruker IP-over-DNS skal håndteres

etter de er merket ikke er tenkt belyst. Det kan kanskje være naturlig å sperre dem (på flyplasser/hotspots,) eller la dem fortsette til man får sporet opp maskinen og eieren (internt i bedrifter.)

1.6 Metodikk

For å få undersøkt problemet i tilstrekkelig grad planlegges det å i stor grad arbeide eksperimentelt. Etter å ha gjort forundersøkelsene bør et sett med teknikker for deteksjon vurderes.

På forhånd har metoder som å beregne mengde DNS-trafikk per klient, samt å finne Kolmogorov-kompleksiteten til DNS-trafikken (nærmere beskrevet senere) kommet opp som mulige måter å løse problemet på.
Den første går ut på at klientadresser som kjører IP-over-DNS vil ha langt mer DNS-trafikk enn vanlige klienter. Kolmogorov-kompleksitet er et matematisk eller algoritmisk begrep som går på hvor mye informasjon som er i DNS-pakkene. Forenklet kan man si hvor mye data som trengs for å representere innholdet. En idé er at de kodede IP-pakkene i DNS-forespørslene vil ha større informasjonsmengde enn vanlige forespørsler.

Hvorvidt disse to ideene stemmer og kan brukes til å påvise IP-over-DNS i trafikk skal undersøkes.

1.7 Relatert arbeid

Denne oppgaven er en viss grad basert på det Nussbaum m.fl. beskriver i konklusjonen i [NNR09]. De skriver:

> From a network administrator point of view, it seems difficult to block TUNS without also blocking legitimate traffic: the only solution left is to reduce the bandwidth of the covert channel by using traffic shaping techniques (to rate-limit the DNS queries), thus making the channel mostly useless.

Løst oversatt sier de at med sin IP-over-DNS-implementasjon "TUNS" er det nesten bare båndbreddebegrensning av DNS-trafikk per klientadresse som står igjen som mulig teknikk for å begrense bruk av den. Hvorvidt dette stemmer skal undersøkes.

1.8 Rapportens oppbygning

Rapporten er delt i fem kapitler. Kapittel 2 inneholder bakgrunnsinformasjon om hvordan DNS fungerer, fulgt av kapittel 3 som forklarer IP-over-DNS i detalj. I kapittel 4 diskuteres ulike metoder for å påvise IP over DNS i DNS-trafikk, og resultatene av utprøving av disse belyses i kapittel 5. Til slutt oppsummeres arbeidet i kapittel 6.

Kapittel 2

Domain Name System (DNS)

2.1 Introduksjon

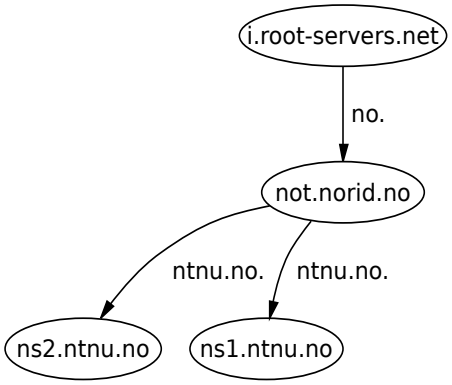
Domain Name System (DNS) er en hierarkisk distribuert database lagd for høy skalerbarhet og kort oppslagstid. Databasen inneholder koblingen mellom navn og IP-adresser på Internett. DNS er en moden protokoll som ble standardisert på midten av 1980-tallet [Moc87a, Moc87b].

De viktigste rollene eller funksjonene i DNS er:

- Autoritative navnetjenere. Disse inneholder (deler av) databasen DNS er. En større organisasjon vil vanligvis av pålitelighetshensyn kjøre minst to autoritative tjenere. Mindre organisasjoner vil vanligvis kjøpe tjenesten hos en annen leverandør (I Norge: Domeneshop, Telenor, andre.)
- Klientprogramvare for oppslag. På engelsk kalt "stub resolvers". Dette er programvare som kjører i operativsystemet og som tilbyr et felles grensesnitt (API) der applikasjoner kan få slått opp navn i DNS.
- Rekurserende navnetjenere. Disse serverene brukes av "stub resolvers" og gjør oppslag mot ulike autoritative tjenere på vegne av dem. Disse tjenerene mellomlagrer (cacher) det de har slått opp en tid, slik at om flere klientmaskiner stiller samme spørsmål så vil påfølgende bli besvart fra cache.

Nivåene i DNS-databasen kalles ofte for en sone (engelsk: zone), eller et domene om man er på de øverste nivåene. I en sone kan man gi vekk myndighet til spesifikke undersoner til andre navnetjenere. Dette kalles delegering. På toppen finner man rootsonen ".". I denne sonen vil det fin-

nes delegering av navnerommet ".no" til de norske nasjonale autoritative navnetjenerene. De norske navnetjenerene vil så for eksempel delegere ansvar for navnerommet (sonen) "ntnu.no" til universitetet sine navnetjenere, eller "telenor.no" til Telenor sine. Det er ofte en mindre kostnad, betalt til nivået over i hierarkiet, involvert i å få tilgang på en del av navnerommet. Et eksempel på dette ser man i figur 2.1, der rootnavnetjeneren *i.root-servers.net* delegerer ".no" til (blant annet) *not.norid.no*, som delegerer sonen "ntnu.no" til de autoritative navnetjenerene *ns1.ntnu.no* og *ns2.ntnu.no*.



Figur 2.1: Delegeringstre for ntnu.no

Initiativet i DNS-protokollen føres vanligvis av en applikasjon. Denne kaller sin stub resolver, som spør rekurserende navnetjener, som spør null eller flere autoritative navnetjenere før svar eller feilmelding sendes tilbake til applikasjonen. I noen få tilfeller kan kommunikasjon bli initiert av serverene direkte, for eksempel når en autoritativ tjener henter en oppdatert kopi av en sone fra en annen autoritativ server.

2.2 Databaseinnslag

Innslag eller radene i en DNS-sone kalles "resource records" (RR). Det finnes ingen vanlig oversettelse, så begrepet "DNS-innslag" blir brukt her. DNS-innslag har følgende attributter:

- Et navn, eller oppslagsnøkkelen.

- En type som blant annet brukes til å si om dette er navn-til-IP-adresse, eller IP-adresse-til-navn.
- Et datafelt som inneholder svaret på oppslagsnøkkelen. Innholdet er avhengig av hva slags type innslag det er.
- Administrative felt som Time To Live (TTL), som sier hvor lenge innslaget kan caches av andre servere.
- Informasjon om hvilken klasse DNS-innslag det er. Ikke tatt i bruk i særlig grad, og satt til "IN" for Internett.

Det finnes en rekke typer DNS-innslag, hvor de viktigste i denne sammenhengen er:

- A. Navn til IP -innslag.
- NS. Delegering av myndighet.
- MX. Kortform for "Mail eXchange" som oppgir hvem som håndterer epost for sonen.
- CNAME. En peker fra ett navn til et annet navn.
- TXT. Innslag som inneholder uspesifisert tekst av fritt ønske. Lite brukt.
- NULL. Et nullinnslag for eksperimentell bruk. Sjelden eller aldri sett i vanlig bruk.

Det finnes en rekke andre typer innslag som SOA, PTR, AAAA og SRV i vanlig bruk, men disse er ikke så relevante her.

Et eksempel på et DNS-innslag vises i 2.1. Her er det gjort et oppslag etter navn-til-IP-koblingen (type A) for "www.vg.no".

Felt	**Verdi**
Navn	www (.vg.no)
Type	A
Svardata	195.88.54.16
TTL	298
Klasse	IN

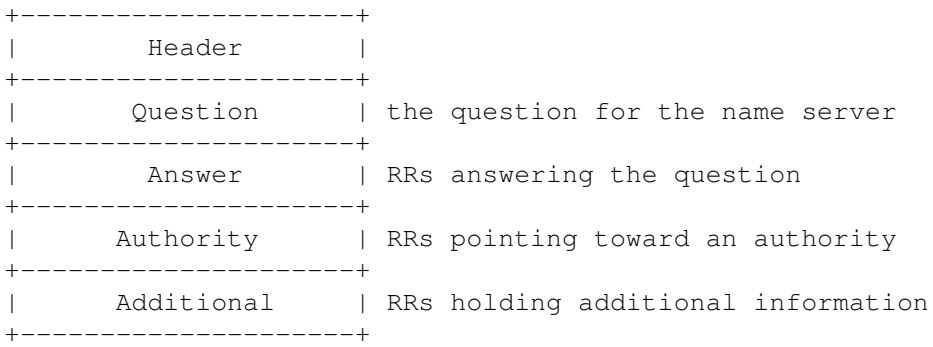
Tabell 2.1: Eksempel på et DNS-innslag (resource record)

2.3 Grammatikk og pakkeformat

Når det kommer til selve grammatikken i DNS, eller protokollprimitivene om man vil, er dette veldig rett frem. Det finnes en pakkeform (engelsk:

on-the-wire format), vist i figur 2.2, som brukes til alle operasjoner.

Ulike felt i DNS-pakken settes basert på om det er en spørring eller et svar. Feltene for svar er ledige i en spørring, mens svarpakker inneholder kopi av spørringen. I tillegg til informasjon om spørringen og eventuelt svar, ligger det med informasjon om hvilke navnetjenere som er autoritative for dette svaret, samt eventuelle ekstra informasjon svarende tjener tror spørrende klient trenger.



Figur 2.2: Pakkeformatet til DNS slik vist i [Moc87b]

Det kan være greit å bemerke seg at DNS-pakkene har ett felt med plass til en oppslagsnøkkel (ett DNS-innslag), men tillater at svar består av flere DNS-innslag. Dette utnyttes i IP-over-DNS.

2.4 Størrelser i DNS

En oppslagsnøkkel kan maksimalt, sonennavn inkludert, være 255 oktetter lang. Denne består av flere ”labels“ som maksimalt kan være 63 oktetter. Labels blir skilt med punktum og danner til slutt et DNS-navn. Som et eksempel består DNS-navnet ”www.vg.no“ av de tre “labels“; ”www“,”vg“ og ”no“.

Pakkestørrelsen i opprinnelig spesifikasjonen av DNS over UDP er satt til 512 oktetter. Om DNS-pakken er større enn det, skal avsenderen forsøke å opprette en TCP-forbindelse i stedet. I praksis er de fleste DNS-pakker mindre enn 512 byte, og TCP er frivillig å støtte på tjenersiden. Dette er en viktig begrensning og forklares bedre i neste kapittel.

Kapittel 3

Tunnellering av IP over DNS

3.1 Teknisk beskrivelse

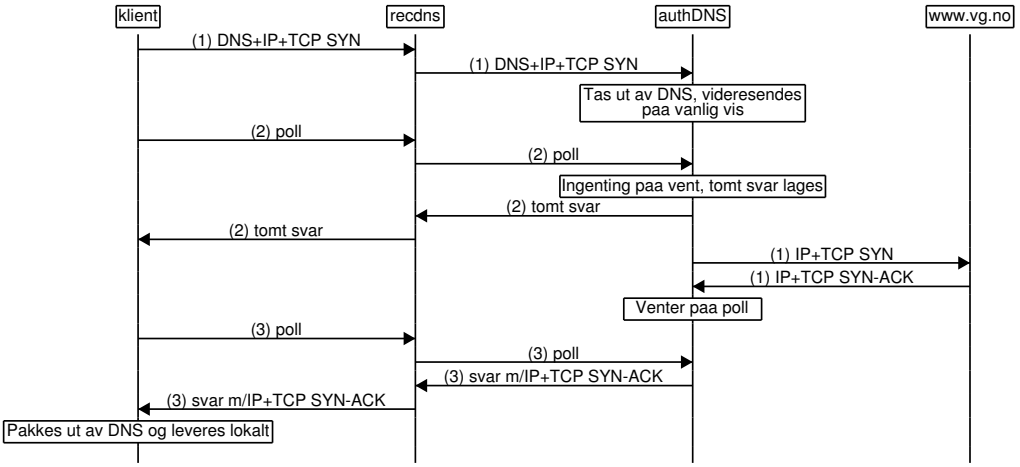
Et vanlig oppsett for IP-over-DNS består av en DNS-klient, en rekurserende navnetjener og en autoritativ navnetjener. Før man kan ta i bruk tunnellen trenger man å sette opp IP-over-DNS -programvaren på en server ute på Internett, og delegere en DNS-sone til den.

For å få opp kommunikasjon sender klienten forespørsler til den rekurserende navnetjeneren med jevn takt. Utgående IP-pakker blir kodet på mest mulig kompakt vis i selve navnet som blir spurt etter, og returpakkene blir kodet i svaret på denne spørringen. Klienten vil sende spørringer selv om den ikke har noen utgående data, slik at serveren kan sende eventuelle data den har liggende i kø.

Et meldingssekvensdiagram (MSC) av en TCP-oppkobling mot www.vg.no er vist i figur 3.1. Klienten spør sin lokale (rekurserende) DNS-server, som videresendes til autoritativ server, som pakker ut IP-pakken og videresender den ut på internett med seg selv om avsender. Ofte benyttes IPv4 NAT her, slik at flere klienter kan skjule seg bak en IP-adresse. Svarpakken kommer tilbake, og pakkes inn. Etter en stund poller klienten og får IP-pakken enkodet i svaret på poll-forespørselen.

3.2 Eksisterende implementasjoner

Iodine er en vedlikeholdt implementasjon av IP over DNS med fri kildekode. Man kan laste ned programvaren fra `http://code.kryo.se/iodine/`. Iodine støtter alle de vanlige måtene



Figur 3.1: MSC med pakkeflyt IP-over-DNS

å kommunisere med IP-over-DNS, og tilpasser seg automatisk til forholdene den er på. Versjon 0.6.0rc1 av Iodine er siste nåværende versjon og den som brukes i dette arbeidet.

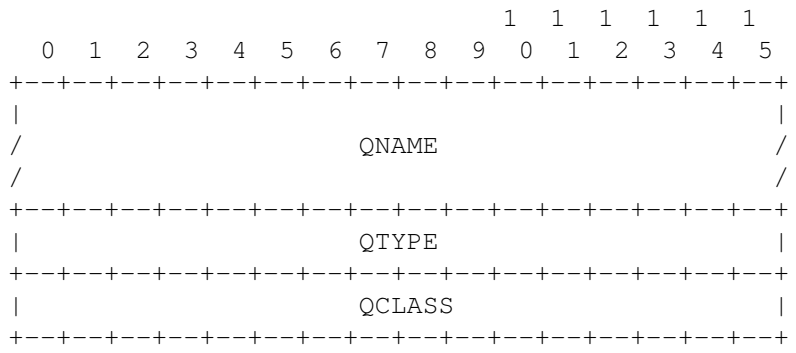
TUNS ble introdusert i [NNR09]. I dette paperet blir de ulike enkodingsmekanismene og metodene til Iodine 0.4.2 og NSTX 1.1beta6 gått gjennom, og forfatteren tar for seg hvorfor disse ikke fungerer alltid. Målet med TUNS var å verifisere at mange kan oppnå bedre resultater gjennom å sende forespørsler strengt i tråd med DNS-RFCene, og kun bruke de vanligste DNS-innslagstypene (A og CNAME). I følge tallene i paperet ser dette ut til å stemme.

NSTX, akronym for "Nameserver Transfer Protocol", er lagd av Florian Heinz og Julien Oster. Dette er den første kjente implementasjonen av IP-over-DNS, offentliggjort i 2000. Målet den gang ble fremsatt som å kunne få gratis Internett-tilgang gjennom Microsoft sin oppringttjeneste.

3.3 Teoretisk bredde på transportkanal i DNS

3.3.1 Utgående

En DNS-spørring består av DNS-headeren (12 byte) og en instans av datastrukturen for "Question" slik beskrevet i figur 3.2. Som man kan se ut av figuren er "QTYPE" og "QCLASS' totalt 4 byte, mens "QNAME" er inntil 255 byte lang. Dette gir plass til fire "labels" av maksimalstørrelsen på 63 oktetter, pluss fire label-headere på 1 byte. **Totalt gir det 252 byte med nyttelast i en maksimal spørrepakke på 259 byte.**



Figur 3.2: Format på Question-delen av en DNS-pakke slik vist i [Moc87b]

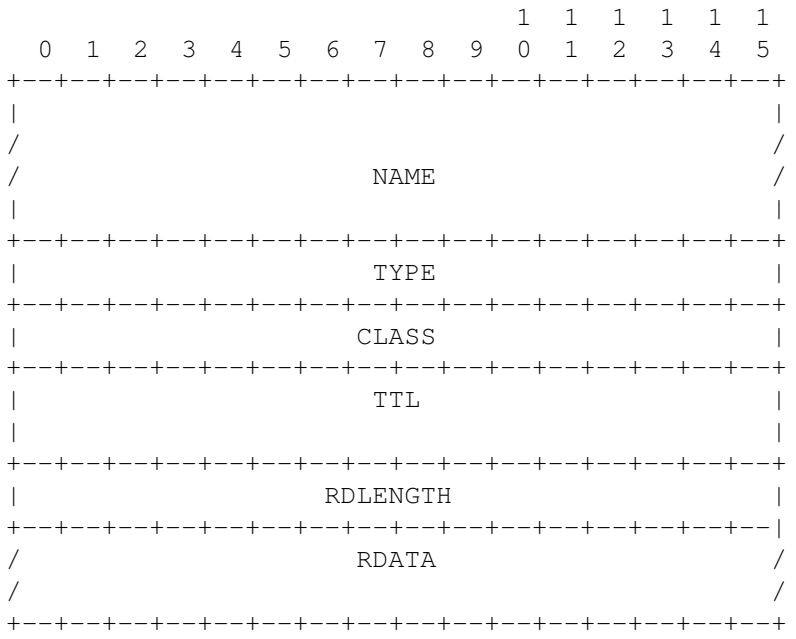
3.3.2 Innkommende

Bredden på innkommende kanal er litt mer flytende enn utgående. Det er mulig å legge informasjon i en til mange "Answer"-strukturer i pakken. Denne datastrukturen er vist i figur 3.3 og tidligere forklart i delkapittel 2.2, hvor headeren ble utelatt i forsøk på å gjøre teksten mer leselig.

Om man fremdeles antar 512 byte UDP-PDU er maksimalt, altså minimumskravet:

- DNS-header: 12 oktetter
- Opprinnelig "question"-struktur, antatt 259 oktetter.
- En "answer"-struct. 2 oktetter referanse til QNAME, 2 oktetter QTYPE, 2 oktetter QCLASS, 4 oktetter TTL, 2 RDLENGTH. Resten RDATA.

Totalt er det 283 oktetter med data i svar-pakken, uten at nyttelast er tatt med.



Figur 3.3: Format på Answer, Authority og Additional -delene av en DNS-pakke slik vist i [Moc87b]

Dette gir 512-283=229 oktetter igjen til innholdet i RDATA, som formateres på ulikt vis basert på hva slags QTYPE som er satt. NULL-typen er definert som ”65535 octets or less“, uten krav til noen spesiell formatering. CNAME-typen skal være gyldig DNS-peker. Andre typer har andre krav, men disse er de to mest relevante for IP-over-DNS.

Om typen er NULL har man 229 oktetter med nyttelast. Om typen er CNAME, må dataene deles opp i 4 labels og man har 225 oktetter tilgjengelig.

3.3.3 Innkommende med EDNS0

EDNS0 ble definert i [Vix99] som ble utgitt i 1999. **EDNS0 fjerner maksgrensen på 512 byte per pakke**, og utvider størrelsen på DNS-headeren. Det er en utvidelse som i hovedsak ble lagd for å kunne støtte sikkerhetsmekanismene i DNSSEC.

Sett utenfra er EDNS0 fullt bakoverkompatibel, og klienter signaliserer støtte ovenfor servere gjennom å legge til et ekstra tomt DNS-innslag

i "Additional"-delen av spørrepakken. Denne ekstra posten er av type OPT og er kun tillatt når en forespørsel sendes over nett. Servere som ikke støtter EDNS0 vil ignorere den og sende et ordinært DNS-svar tilbake. Servere som støtter EDNS0 vil fylle de ekstra feltene i OPT-posten i returpakken.

I sammenhengen med IP-over-DNS er det primært den økte pakkestørrelsen som er viktig. Om alle involverte servere støtter EDNS0 kan pakkestørrelsen økes til MTU på linkene, vanligvis opptil 1500 byte.

3.3.4 Koding

Teksten i DNS-pakker er kodet i et subsett av tegnsettet US-ASCII. Vanlige bokstaver, tall og bindestrek er lov. De fleste implementasjoner i dag skiller (og transporterer urørt) store og små bokstaver, men i henhold til spesifikasjonen er dette valgfritt. Man har altså et sted mellom 37 (a-z, 0-9, -) og 63 (a-z, A-Z, 0-9, -) ulike symboler tilgjengelig. Enkelte andre tegn, som _ (understrek) er i bruk, men offisielt ikke godtatt.

Med 37 tegn kan man bruke Base32 som kodingsmekanisme (5 bit per byte i pakken), og med 64 tegn kan man bruke Base64 (6 bit per byte i pakken) som kodingsmekanisme.

3.3.5 Estimert båndbredde

Om man tenker seg en ideell kanal uten tap, der man sender en DNS-spørring per sekund og svaret kommer tilbake med en gang. Denne kanalen har en kapasitet på 225 oktetter inngående per sekund (uten EDNS0.)

En vanlig rekurserende navnetjener tar i dag unna opp i mot 40000 spørringer/sekund [Hub10]. Dette inkluderer en god del cachede innslag, så man kan konservativt si at 4000 spørringer/sekund med IP-over-DNS -trafikk bør være overkommelig. Dette gir en teoretisk overføringshastighet på $4000 * 225 = 900\,000 \frac{byte}{sekund}$ inn og ut.
Om man bruker base32 som kodingsmekanisme, med 225 oktetter per sekund tilgjengelig får man $5 * 225 = 1125 \frac{bit}{sek}$ binær nyttelast. I denne datamengden kan $\frac{1125 bit}{8 \frac{bit}{byte}} \approx 140\ byte$ med binærdata transporteres i hver DNS-forespørsel.

Med 4000 pakker per sekund blir det 560 KiB/s overføringskapasitet inn og ut, fremdeles uten å bruke de mer kompakte kodingsformene eller å bruke den økte pakkestørrelsen EDNS0 åpner for.

Disse tallene er ikke helt rett ettersom forhold som pakketap, sjekksumfeil og køing i ulike servere ikke blir vurdert. De gir dog en god indikasjon på det underliggende poenget; **den skjulte kommunikasjonskanalen er i stor grad brukbar til å transportere informasjon over.**

I [TvLL08] fra 2008 hevder van Leijenhorst m.fl. at en slik kanal kan ha inntil 110KiB/s båndbredde. Dessverre har de utelukkende sett på NSTX og OzymanDNS, som begge er av historisk art, og konklusjonene deres er derfor ikke lengre så relevante.

Kapittel 4

Teknikker for deteksjon av IP-over-DNS

4.1 Dagens tilstand

Tilstanden på deteksjon av IP-over-DNS i dag er at det ikke finnes noe publisert innenfor temaet som forfatteren har vært i stand til å finne.

Det finnes noen blogg-poster på Internett, primært `http://blog.vodun.org/2006/12/thoughts-on-traffic-analysis-for.html`, `http://blog.vorant.com/2006/05/traffic-analysis-approach-to-detecting.html` samt `http://www.daemon.be/maarten/dnstunnel.html`. Disse beskriver ideene for å måle Kolmogorov-kompleksitet samt forsøke å gi beskjed om en klient sender mistenkelig mye DNS-data, men de har ikke undersøkt videre eller prøvd dem ut i praksis.

4.2 Overordnet om problemet

Å detektere et signal blant mye støy er en ganske vanlig problemstilling innen signalbehandling. Det finnes mange etablerte teknikker for å gjøre dette, så fremt man får konkretisert hvordan signalet sin oppførsel er. I denne sammenhengen kan man se på en IP-over-DNS -klient sin trafikk som signalet og alle andre klienter som støy.
Den ideelle deteksjonsteknikken vil:

- Kunne detektere IP-over-DNS med 100% sannsynlighet.
- Aldri feilklassifisere vanlig trafikk som IP-over-DNS.

- Utføre klassifikasjonen i løpet av tilnærmet ingen tid, slik at kanalen aldri er brukbar for trafikk.

Utfordringen er å klare å kvantifisere disse tre målene i teknikkene beskrevet under, og på en god måte kunne si hvilke metoder som passer best i ulike situasjoner.

4.2.1 Reaksjonstid

Deteksjon av avvik ønsker man gjerne at skal skje så kjapt som mulig, slik at tiltak kan gjøres. Dessverre er det en slags motsigelse gjennom hvor nøyaktig man kan si noe om en klient/oppførsel, og hvor lang tid man har observert den. Fra sannsynlighetslæren kan man se på Bernoulli-forsøk; om man har en 90% sikker test på positiv bruk av IP-over-DNS, som krever 10 sekunder med data for å gjøre sin beregning. Ved å kjøre denne flere ganger på etterfølgende datasett (20,30,40 sekunder) vil man få en binomialfordelt sannsynlighetsfordeling og bli mer og mer sikker på beslutningen sin.

I dette arbeidet har man lagt seg på at 10-30 sekunder er akseptabelt. Målet her er å se på ulike teknikker og få kartlagt dem, ikke å få optimalisert en teknikk så bra som den kan bli.

4.2.2 Falske positiver

Falske positiver betegner feilklassifisering av klienter som kjører IP-over-DNS. Ettersom DNS er så grunnleggende for funksjonen av en Internettforbindelse, vil en eventuell feilklassifisering med påfølgende sperring fjerne alle muligheter for en vanlig bruker å benytte seg av tjenesten. I et hotspot-miljø vil dette være spesielt viktig, ettersom brukeren nylig har betalt for tjenesten som ikke lengre virker.

I en bedriftssammenheng er dette et litt mindre viktig moment, så fremt feilklassifiseringen ikke skjer så ofte at det blir en administrativ byrde for de/den som må undersøke saken.

4.3 Avviksdeteksjon

Avviksdeteksjon, eller "anomaly detection", er et bredt fagfelt med mange publiseringer på. Slike avvik kan for eksempel være rar oppførsel av

klientmaskiner etter datainnbrudd, eller uvanlig bruk av kredittkort ifm. svindel. En rekke ulike grupperinger og teknikker er beskrevet i [CBK09]. En vanlig klassifisering av teknikker for avviksdeteksjon er:

- Supervised. Teknikken trenes med kjent avvikstrafikk og kjent vanlig trafikk.
- Semi-supervised. Teknikken trenes med kjent vanlig trafikk. Det lages en modell og senere nyttetrafikk blir vurdert opp mot hvor godt den passer med modellen.
- Unsupervised. Teknikken gis en masse ikke-konkretisert trafikk der mesteparten er vanlig.

I dette arbeidet har man primært benyttet "supervised", men enkelte metoder kan nok ses på som "semi-supervised" også, spesielt der statistiske modeller er tatt i bruk.

4.4 Dimensjonsanalyse av DNS-trafikk

Et grunnleggende tema før ulike algoritmer tas i bruk er å redusere antall variabler. Til denne bruken vil med sannsynlighet en rå dump av DNS-trafikk sine IP-pakker være inndata. Dette settet vil kunne beskrives på mange ulike måter.

Temporalt syn:

- spørringer per tidsenhet. (qps)
- byte/oktetter per tidsenhet.
- tid mellom hver spørring.
- pakkestørrelse
- entropi-innhold

Disse vil i ulik grad kunne bestemmes for hele datasettet, for en enkelt klient eller for et sett med klienter.

Man kan videre tenke seg et spektralt/frekvensbasert syn, der en samlet last av spørringer består av n strømmer til m mottakerdomener.

4.5 Punktprøving

Det er ulike måter å sample, eller på norsk punktprøve, datasettene. I denne oppgaven er uniform og hendelsesbasert sampling benyttet.

Uniform sampling vil si at det produseres et sample (datasett) med faste tidspunkter, uavhengig av aktiviteten i "systemet". Pseudokode (i python) for dette er vis i 4.1. Vanligvis er 10-30 sekunder benyttet. **Hendelsesbasert** punktprøving tar i stedet utgangspunkt i når noe har skjedd i systemet. Her er det hendelsen "svar sendes til klient" som er brukt, og et sample dannes når så så mange slike hendelser er skjedd. Dette er illustrert med pseudokoden i 4.2.

I uttestingen av teknikkene benyttes disse til å dele opp filene fra tcpdump (PCAP-filer) i tidsintervaller. Dette gjør det enklere å se hvordan metrikkene utvikler seg over tid.

I forhold til deteksjonstid er det antatt at hendelsesbasert sampling vil kunne gi best signal/støy -forhold når vi vet at IP-over-DNS -klienter sender over gjennomsnittet mange pakker og mengde data. En slik samplemetode vil kunne reagere raskt på "tunge" klienter, samtidig som den uhindret vil kunne analysere klienter som sender sakte. Klienter som sender lite trafikk kan være legitime klienter, eller IP-over-DNS -klienter som forsøker å holde seg skjult.

Figur 4.1: Pseudokode for uniform sampling

```
tid_forrige_sample = None
sampletid = 30 # sekunder
buf = []
for svar in alle_dnssvar_i_datafil:
    buf.append( svar )
    if not tid_forrige_sample:
        # finne starttidspunkt
        starttid = svar.tidspunkt
        continue

    else:
        if tid_forrige_sample + sampletid <= svar.tidspunkt:
            beregn( buf )
            tid_forrige_sample = svar.tidspunkt
            buf = []
```

Figur 4.2: Pseudokode for hendelsesbasert sampling

```
klienter = {}
for svar in alle_dnssvar_i_datafil:
    if not klienter.has_key( svar.ip ):
        klienter[ svar.ip ] = [ ]
    klienter[svar.ip].append( svar )

    if len(klienter[svar.ip]) >= numpackets:
        beregn( klienter[svar.ip] )
        klienter[svar.ip] = []
```

4.6 Statistiske modeller for Internett-trafikk

I tradisjonelle telekommunikasjonssystemer kan man modellere ankomst med en Poissonprosess. Dette er matematisk behagelig, og åpner for velbeskrevede teknikker for analyse. Mange Poissonprosesser som legges sammen blir en ny Poissonprosess med summert intensitet.

Internett-trafikk oppfører seg litt annerledes. Man anser at mange-til-mange -basert internett-trafikk har fraktalske egenskaper, eller utøver "self-similarity" [PF95]. Dette betyr at det generelle trafikkmønsteret ikke endrer utseende selv om man går ut eller inn en eller flere størrelsesordner. Trafikktoppene flates ikke ut, men blir desto høyere når trafikken øker. Generelt sett kan man ikke anta at tiden mellom pakker er negativt eksponentialfordelt, slik man kan i en Poissonprosess.

Dette generelle mønsteret gjør modelleringsjobben for de teknikkene der man bygger en felles modell litt mindre rett frem.

4.7 Kolmogorovkompleksitet

Kolmogorovkompleksitet er et begrep utviklet av Ray Solomonoff, Andrey Kolmogorov og Gregory Chaitlin på 1960-tallet. Det er et matematisk konsept som sier noe om hvor mye informasjon som finnes i et stykke data.

Et eksempel kan være en enkel tallrekke (Fibonacci): 0, 1, 1, 2, 3, 5, 8, 13... Denne rekken kan beskrives både ved hjelp av de rene tallene, eller enklere ved hjelp av definisjonen av Fibonacci-rekken. $F_n = F_{n-1} + F_{n-2}; F_0 = 0, F_1 = 1$ Om dette faktisk er den definisjonen som krever minst plass, kan dette sies å være en måte å beregne Kolmogorovkompleksiteten til tallrekken.

Det er ikke gitt at det er mulig å entydig definere kompleksiteten til et stykke data, ettersom man ikke har full kunnskap om alle mulige mønstre som kan generere dataene. Dette og en mer analytisk presentasjon av emnet er beskrevet i [GV99].

I [WP05] brukes Wagner og Plattner Kolmogorovkompleksitet til å merke endringer i trafikkmønstre. Deres problem gikk ut på å merke utbredelsen av nye ormer på Internett. De brukte datakomprimering

(med zlib) som et estimat på hvor stor kompleksitet dataene har, og denne metoden virker fornuftig å benytte seg av.

4.8 Regelbaserte metoder

Regelbaserte systemer for å kjenne igjen IP-over-DNS finnes. Eksempler på dette er open-source løsningen Snort, og kommersielle leverandører som ISS/IBM. De virker gjennom å la forhåndsskrevne regler klassifisere alle data som går gjennom filteret, og gir alarmer om oppgitte data blir observert.

Regelmotorer kan være svært effektive til å kjenne igjen kjente angrep. Hovedproblemet er at angrepene må være kjent på forhånd, og noen må ha tatt seg tid til å skrive regelen samt få den distribuert til alle involverte.
Som eksempel er Snort sine uoffisielle regler for å legge merke til Iodine-trafikk vist i figur 4.1. Dette regelsettet tolkes som et sett med OG-ede kriterier. Alle må stemme for at regelen skal treffe. Regelen for spørringer tolkes slik:

- Spørringer må gå mot en DNS-server som lytter på UDP/53.
- content -linjen med “01 00 [..]“ kjenner igjen headeren på DNS-spørringer.
- content -linjen med “00 00 29 01 [..]“ kjenner igjen en EDNS0-pakke med inntil 4096 byte innhold, men inget faktisk innhold.

På lignende vis tolkes regelen for svarpakke slik:

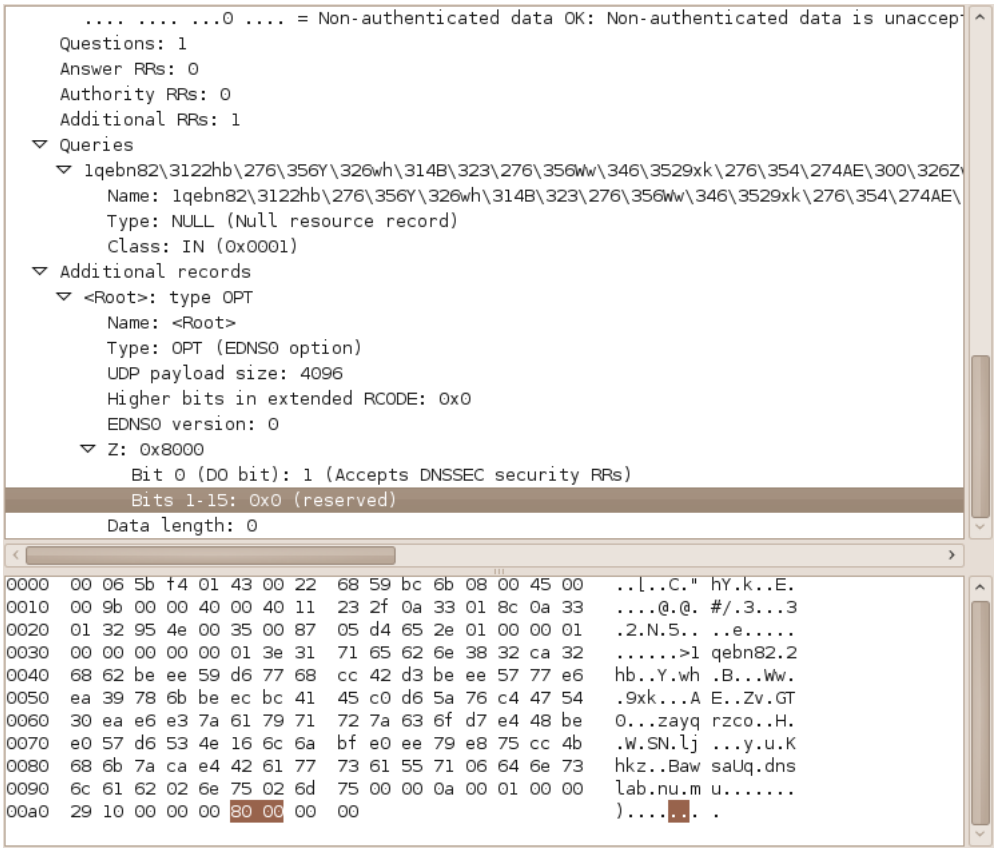
- Pakken må komme fra UDP/53.
- content -linjen med “01 00 [..]“ kjenner igjen headeren til et generelt DNS-svar.
- content -linjen med “00 00 0a 00 01“ treffer siste del av iodine-pakken. Den er dessverre ikke korrekt, og den første 00 -sekvensen er de to siste heksadesimale verdiene i svarpakken. De resterende ”00 0a 00 01“ treffer at Iodine benytter QTYPE NULL, som ellers svært sjelden er i bruk.

Den enkleste metoden for å omgå en regelmotor er å finne et eksempel på hvordan den fungerer, og så modifisere angrepet til regelen ikke lengre treffer.

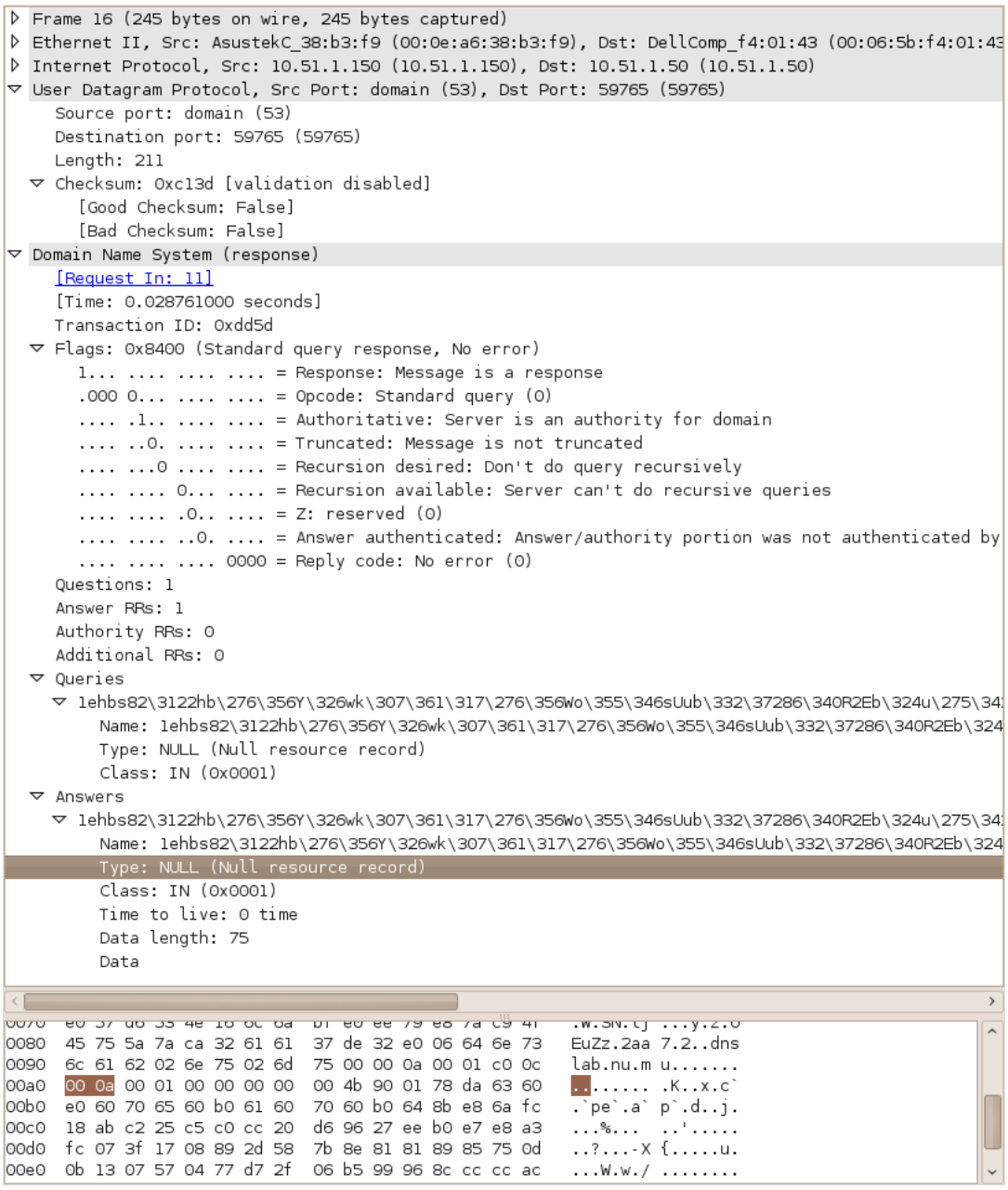
```
# detects iodine covert tunnels (over DNS),
# send feedback on rules to merc [at] securitywire.com
alert udp any any -> any 53 \
  (content:"|01 00 00 01 00 00 00 00 00 01|"; \
  offset: 2; depth: 10; \
  content:"|00 00 29 10 00 00 00 80 00 00 00|";  \
  msg: "covert iodine tunnel request"; \
  threshold: type limit, track by_src, count 1, seconds 300; \
  sid: 5619500; rev: 1;)

alert udp any 53 -> any any \
  (content: "|84 00 00 01 00 01 00 00 00 00|"; \
  offset: 2; depth: 10; \
  content:"|00 00 0a 00 01|";  \
  msg: "covert iodine tunnel response"; \
  threshold: type limit, track by_src, count 1, seconds 300; \
  sid: 5619501; rev: 1;)
```

Figur 4.1: Snort-regler for å påvise Iodine-trafikk



Figur 4.2: Iodine-spørring på pakkenivå



Figur 4.3: Iodine-svar på pakkenivå

Som man kan se av regelene nevnt tidligere, så søker den etter svarpakker med QTYPE NULL. For mellomliggende servere har denne meldingstypen ingenting å si, det er kun klienten og serveren som bryr seg. Det er fullt mulig å endre på programkoden til Iodine slik at en annen meldingstype blir brukt.

Dette vil føre til at regelsettet aldri vil treffe, og tunnellen kan brukes ubemerket.

Vurdering

Slik regler kan fungere fint til å kjenne igjen et stort sett med historiske og automatiserte angrep. I enhver sammenheng der man har en intelligent og motivert angriper som har mulighet til å prøve seg frem, vil det stort sett la seg gjøre å unngå dem.

Ut over den overfladiske utprøvingen av snort som ble gjort for dette avsnittet har regelmotorer ikke vært fokusert på i dette arbeidet, og er derfor ikke videre undersøkt.

Kapittel 5

Resultater

5.1 Forhåndsanalyse av datasett

For å få dypere innsikt i hvordan DNS-trafikk opptrer ble tilgjengelige datasett undersøkt. Målet var å få en følelse med antall spørringer per klient, i hvor stor grad klienter kommer og går, og hvordan fordelingen mellom tunge og lette klienter var.

Disse målingene er alle tatt med tcpdump på en rekurserende navnetjener. All DNS-trafikk er tatt opp, men for å gjøre behandlingen enklere er det kun DNS-svar fra rekurserende til klienten som blir analysert. Grunnen til dette er at både klienten og rekurserende navnetjener kan finne på å sende pakker på nytt om den tror UDP-pakkene er gått tapt eller et tidsavbrudd skjer. For å slippe å holde orden på hvilke forespørsler som er kopier, benyttes kun svarene tilbake til klienten.

5.1.1 Datasett med generell trafikk

En oversikt over nøkkeltallene fra de generelle pakkedumpene fra NTNU vises i tabell 5.1.

Datasett	Unike klienter	Spørringer	data [byte]	lengde [s]
2010-04-08	3031	127 321	11 080 439	3599
2010-04-09	2694	116 607	9 680 839	3599
2010-04-13	3478	172 576	14 744 949	3599
2010-04-27	3392	139 271	16 306 677	3600
sum		555 775	51 812 904	

Tabell 5.1: Karakteristikker generelle datasett

Oppsummert er gjennomsnittlig pakkestørrelse 92,23 byte. Det er ellers ingen større variasjoner mellom datasettene, og det er ingenting som tyder på feil i målingene.

Det siste datasettet fra 27. april inneholder en (kjent) klient som kjører Iodine 0.6.0rc1. Tunnellen stod stort sett uvirksom de første 20 minuttene, ble aktivt brukt med SSH i 20 minutter, og stod uvirksom de siste 20.

5.1.2 Datasett med IP-over-DNS

I forbindelse med utprøving av de forskjellige kjente implementasjonene av IP-over-DNS ble det tatt pakkedumper av tre situasjoner: ren bakgrunnspolling uten særlig trafikk over tunnellen, aktiv bruk av SSH over tunnellen, og til slutt aktiv webbruk over tunnellen. Disse er oppsummert i tabell 5.2. Forkortelsene som brukes er:

- MTBQ: "mean time between queries". Snittid mellom hver spørring.
- MPS: "mean packet size". Gjennomsnittlig pakkestørrelse.
- BPS: "bytes per second". Justert etter lengde på datasettet.
- QPS: "queries per second". Justert etter lengde på datasettet.

Trafikk-miks	**Programvare**	**MTBQ** [s]	**MPS [B]**	**BPS** [B/s]	**QPS** [Q/s]
Kun polling	iodine 0.6.0rc1	2.80	127.27	50	0.39
	NSTX (MTU 1500)	0.49	108.00	220	2.04
	TUNS	1.51	103.94	70	0.68
SSH	iodine 0.6.0rc1	0.11	382.17	3407	8.92
	NSTX (MTU 1500)	0.44	246.42	560	2.27
	NSTX (MTU 500)	0.38	283.71	749	2.64
	TUNS	0.11	208.92	1841	8.81
Webtrafikk	iodine 0.6.0rc1	0.03	782.91	22635	28.91
	NSTX (MTU 500)	0.02	295.23	12782	43.30
	TUNS	0.01	221.83	19782	89.18

Tabell 5.2: Karakteristikker spesielle datasett

NSTX har problemer med pakketap om man ikke skrur ned "Maximum Transfer Unit" (MTU) til 500, dette ble merket først etter at SSH-sesjonen hengte seg opp. Fra tabellen kan man ellers lese:

- Pakkestørrelsen er alltid større enn gjennomsnittet på 92 byte nevnt tidligere.
- Klientene sender DNS-forespørsler hyppig.
- Beregnet MTBQ er ganske lik inversen av pakketakt QPS, slik den bør være.

5.2 Variabler til deteksjonsmetodene

Metodene som er utviklet er lagd for å ta inn en enkelt parameterliste, beskrevet i tabell 5.3, både for kjøring i sanntid og for behandling av PCAP-filer. Dette gjør det enklere å kvantifisere de ulike innverdiene og kunne variere dem fritt. Noen av variablene blir først forklart senere i teksten, og er tatt med her for oversikten sin del.

Variabelnavn	**Beskrivelse**
numpackets	Pakker per sample.
domain_max_percent	Prosentfaktor på pakker per mottakerdomene i et sample.
movingaveragedepth	Antall samples må gi advarsel før alarm gis.
bps_minimum	Minimum mengde data per sekund per klient i et sample.
qps_minimum	Minimum spørringer per sekund per klient i et sample.
mps_minimum	Minimum gjennomsnittlig pakkestørrelse i et sample.
kolmo_minimum	Minimum Kolmogorovrate før alarm gis

Tabell 5.3: Deteksjonsvariabler

5.3 Deteksjon gjennom båndbreddeforbruk

En klient som benytter IP-over-DNS vil over tid sende langt mer data gjennom DNS-serveren enn en vanlig klient. En annen forutsetning er at vanlige DNS-klienter vil sende noen få forespørsler, og så være stille en relativt lang stund. Man kan se for seg at en IP-over-DNS -klient må polle etter nye data, og sannsynligvis sende en del data underveis, og vil derfor skille seg ut.

Om disse to antagelsene holder, vil man kunne lage en profil på hva som er vanlige mengder data å overføre for en klient, og så si at de klientene som er over 3 standardavvik over snittet er mistenkelige.

En mulig kritikk mot dette er at om "søvnperioden" til klienter er lengre enn sampleperioden, vil man ikke få dem med i beregningen og beregningen vil være skjev oppover. Dette bør kunne håndteres gjennom å øke tidsperioden et sample går over, på bekostning av reaksjonstid.

Ettersom enkelte miljøer vil ha servere som gjør mange DNS-oppslag bør en eventuell implementasjon i faktisk bruk inneholde mulighet for hvitelisting.

5.3.1 Første forsøk

Punktprøvingslengden settes til 10 sekunder. Kun data fra ett sample vurderes av gangen. Resultatene er oppsummert i tabell 5.4.

Datasett	Klienter med alarm	Falske positiver
2010-04-08	269 av 0	269
2010-04-09	272 av 0	272
2010-04-13	285 av 0	285
2010-04-27	84 av 1	83

Tabell 5.4: Resultater første forsøk båndbredde

Kommentarer

Dette første forsøket gav uholdbart mange falske positiver. Miksen av disse er alt fra tunge klienter til ca. alle epostservere på NTNU.

5.3.2 Andre forsøk

Antagelsen om at DNS-trafikk har transienter ("bursty") ble tatt i bruk. Utvalget og sample-lengde er som før, men for at alarmen skal gå må man ha blitt merket også de to forrige samplene. Reaksjonstiden for filteret er derfor minst 30 sekunder. Resultatene fra denne testen er vist i tabell 5.5.

Kommentarer

Virker som implementasjonen som beregner standardavvik og varians er ganske ineffektive, analysen går merkbart treigere enn før. Resultatmessig er dette langt bedre enn den forrige, med mye lavere andel falske positiver. Den kjente IP-over-DNS -klienten 27. april blir korrekt identifisert.

Datasett	Klienter med alarm	Falske positiver
2010-04-08	5 av 0	5
2010-04-09	5 av 0	5
2010-04-13	8 av 0	8
2010-04-27	2 av 1	1

Tabell 5.5: Resultater andre forsøk båndbredde

Om man ser på fordelingen av klienter er det 13 unike IP-adresser som gir disse falske alarmene. Dette er vel innenfor rekkevidden av å kunne hvitelistes. Ut i fra hva disse IP-adressene ser ut til å brukes til, virker det som en del trådløse arbeidsstasjoner er med i listen. Dette er ikke et spesielt lovende trekk med teknikken.

5.3.3 Båndbreddeforbruk oppsummert

Denne teknikken ser ut til å kunne fungere, men gir en god del falske positiver. Det virker derfor mer hensiktsmessig å gå i dybden på andre teknikker i stedet. Det kan godt være at man kan få bedre resultater gjennom å bruke hendelsesbasert punktprøving, men dette er ikke testet ut.

5.4 Deteksjon ved hjelp av Kolmogorov-kompleksitet

Dette var den opprinnelige planen for å detektere IP-over-DNS. Gjennom å bruke hvor stor grad en klient sin trafikk lar seg komprimere som en estimator på Kolmogorov-kompleksitet, undersøk om dette kan brukes til å detektere IP-over-DNS -klienter. Kolmogorovraten er definert slik at data som ikke lar seg komprimere har rate nært 1, mens vanlige data som lar seg komprimere ligger fra 0.5 og nedover.

Denne teknikken basert på arbeidet gjort i [LH07] og [WP05].

5.4.1 Forundersøkelse

Datasettet med en kjent IP-over-DNS -klient (27. april) ble lastet inn og sortert på antall oktetter per klient sendt i løpet av tidsrommet. Datasettet ble så normalisert basert på høyeste observerte verdi for overførte data.

I figur 5.1 er disse plottet. Klientnummer sortert synkende basert på antall byte sendt på horisontal akse og klienten sin Kolmogorovrate på vertikal akse. Kolmogorovraten er beregnet som summen av all data i UDP PDUen komprimert, delt på summen av den ukomprimerte dataen. **Pakkingen skjer per pakke**. Målet med denne grafen var å kunne visuelt undersøke om det er noe åpenbart mønster mellom hvor mye trafikk som sendes og hvor stor kompleksitet denne har.

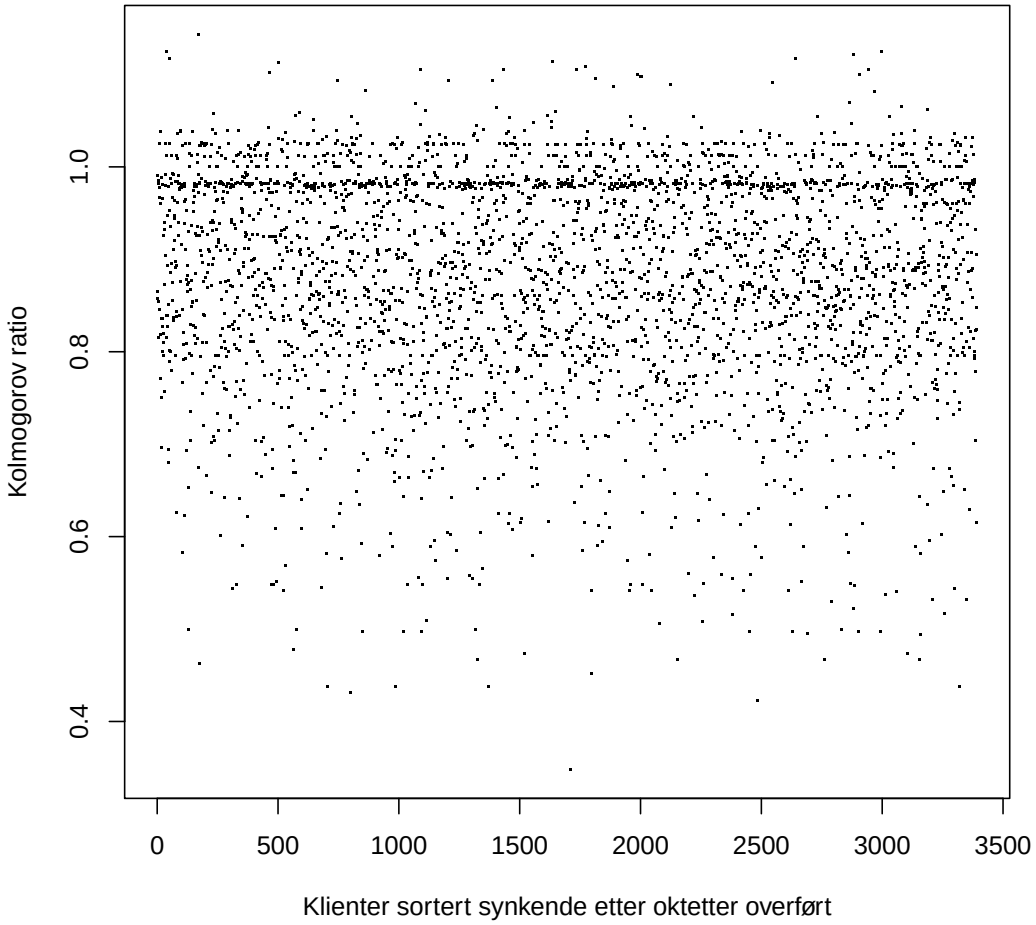
Oppsummert kan man si at på tidsskalaen for hele samplet, er de aller fleste klienter mellom 0.8 og 0.97 i slutt-størrelse etter komprimering. I underliggende rapporter kommer det frem at Iodine-klienten har en samlet komprimeringsrate på 1.01. Pakkene blir i snitt ikke komprimert. Ut i fra hva man kan lese på grafen, gjelder dette også en god andel av andre klienter som er synlige i samplet. Det kan virke som at å komprimere per pakke er en dårlig ide.

Et nytt forsøk gjøres der all data per klient blir aggregert over hele datasettet (en PCAP-fil) og **pakket til slutt**. Denne rutinen ble kjørt på de fire generelle datasettene, og resultatet vises i tabell 5.6.

Datasett	**# unike klienter**	**Gj.snitt K.-rate**	**Std.dev.**
2010-04-08	714	29.0842%	6.78088
2010-04-09	649	28.4357%	6.67313
2010-04-13	885	28.8660%	6.99189
2010-04-27 (alle)	790	29.1495%	7.34973
2010-04-27 (uten kjent IP-over-DNS -klient)	789	29.0823%	7.10806
Kjent IP-over-DNS -klient		82.1331%	

Tabell 5.6: Resultater forundersøkelse Kolmogorov-kompleksitet

Tolkingen av denne tabellen er at Kolmogorovraten er liten for de fleste klienter. Om man tar alle PDU-data over hele filen og pakker det, blir resultatet i gjennomsnitt 28.4 - 29.1% så stort som originaldataene. Som man kan se fra tabellen ligger den kjente klienten på 82.1%. Dette er til tross for at denne kjente IP-over-DNS -klienten har sendt (upakket) 4 829 KiB data i tidsperioden, mens gjennomsnittet er 13003.2 byte. Mer data burde gi bedre komprimeringsgrad, i allefall når hele DNS-pakken er



Figur 5.1: Kolmogorov-kompleksitet som funksjon av oktetter sendt

med slik at headerfelt er repetert mange ganger. Dette er altså ikke tilfellet.

Dette er naturligvis et funn av begrenset nytte, ettersom data for en hel time er tatt med. Om man kan akseptere en times reaksjonstid kan dette brukes.

5.4.2 Praktisk forsøk

Hendelsesbasert punktprøving ble brukt. For å unngå å måtte implementere en hel parser for DNS-pakkeformatet ble en kjapp avgrensning tatt: deler av spørringen og alle svarfelt finnes etter byte 30 i svarpakken. Dette er en grov tilnærming som tar med en del faste headerfelt (som bør komprimeres bra), men virket som en hurtig måte å få evaluert teknikken videre på. I tillegg sammenføyes (konkatineres) dataene fra samtlige pakker i samplet før de kjøres gjennom zlib, slik at den får mer å jobbe med. Pseudokode for beregningen vises i figur 5.1.

Nivået for å si om noe er mistenkelig (minimum_kolmoratio) settes slik at pakket størrelse må være minst 80% av opprinnelig størrelse, i tråd med verdien observert for hele datasettet.

Figur 5.1: Pseudokode for Kolmogorov hendelsesbasert sampling

```
def beregn( liste_med_pakker ):
    buf = ''
    for pakke in liste_med_pakker:
        buf = buf + pakke[30:]
    pakket = zlib.compress( buf )
    kolmoratio = len(pakket) / len( buf )
    return kolmoratio
```

Resultat på spesielle datasett vises i tabell 5.7 og tabell 5.8. Her fremkommer et problem, i praksis at bare Iodine blir oppdaget. Dette er nok en konsekvens av at de andre tunnel-mekanismene kjører med langt lavere pakkestørrelse, slik at å samle numpackets antall pakker total sett blir lite data. Om man går ned til kolmo_minimum på 0.5 får man treff i spesielle datasett på SSH og web-trafikk.

Dessverre viser det seg at når man kjører så lav minimum Kolmogorovrate på de generelle datadumpene, får man svært mange falske positiver. Dette vises i tabell 5.9.

Datasett	num-packets	kolmo-minimum	Positiver	Resp-tid
iodine-ssh	20	0.80	1 av 1 kjente (0 falske, 0 mangler)	86.62s
iodine-web	20	0.80	1 av 1 kjente (0 falske, 0 mangler)	18.01s
nstx-ssh-mtu1500	20	0.80	0 av 1 kjente (0 falske, 1 mangler)	
nstx-ssh-mtu500	20	0.80	0 av 1 kjente (0 falske, 1 mangler)	
nstx-web	20	0.80	0 av 1 kjente (0 falske, 1 mangler)	
tuns-ssh	20	0.80	0 av 1 kjente (0 falske, 1 mangler)	
tuns-web	20	0.80	0 av 1 kjente (0 falske, 1 mangler)	

Tabell 5.7: Spesielle datasett Kolmogorovkompleksitet

Som en liten kuriositet ser man nederst i tabell 5.9 at selv om man øker numpackets til svært høye verdier (og dermed deteksjonstiden) får man ikke vekk de falske positivene. Minimum Kolmogorov-rate på 0.5 er rett og slett for lavt til å brukes praktisk, men fremdeles for høyt til å fange opp alle typer IP-over-DNS. Det ser ut til å fungere fint på Iodine, som sender store fine pakker.

Dette med at Iodine lar seg detektere tyder på at en annen sample-metode som gjør beregner etter n byte i stedet for n pakker ville fungert bedre. Dette er ikke testet ut.

5.4.3 Kolmogorovkompleksitet oppsummert

Kolmogorovkompleksitet ser ut til å kunne fungere bra og raskt i de tilfellene man har nok data tilgjengelig. I tilfellene der man har lite data mister man raskt evnen til å skille mellom vanlige klienter og kjente IP-over-DNS -klienter.

5.5 Deteksjon gjennom data per utgående domene

Premisset bak denne ideen er å gruppere en DNS-klient sine spørringer basert på domenet som spørres etter. Gjeldende implementasjoner av IP-over-DNS fungerer ved at man setter opp et domene ute på Internett og kommuniserer gjennom dette. En klient vil derfor ha svært mange spørringer og sende mye data til og fra en mottaker.

Datasett	num-packets	kolmo-minimum	Positiver	Resp-tid
iodine-ssh	20	0.50	1 av 1 kjente (0 falske, 0 mangler)	2.63s
iodine-web	20	0.50	1 av 1 kjente (0 falske, 0 mangler)	18.01s
nstx-ssh-mtu1500	20	0.50	1 av 1 kjente (0 falske, 0 mangler)	19.04s
nstx-ssh-mtu500	20	0.50	1 av 1 kjente (0 falske, 0 mangler)	11.28s
nstx-web	20	0.50	1 av 1 kjente (0 falske, 0 mangler)	2.24s
tuns-ssh	20	0.50	0 av 1 kjente (0 falske, 1 mangler)	
tuns-web	20	0.50	0 av 1 kjente (0 falske, 1 mangler)	
iodine-ssh	50	0.50	1 av 1 kjente (0 falske, 0 mangler)	4.62s
iodine-web	50	0.50	1 av 1 kjente (0 falske, 0 mangler)	18.17s
nstx-ssh-mtu1500	50	0.50	1 av 1 kjente (0 falske, 0 mangler)	68.49s
nstx-ssh-mtu500	50	0.50	1 av 1 kjente (0 falske, 0 mangler)	36.32s
nstx-web	50	0.50	1 av 1 kjente (0 falske, 0 mangler)	2.25s
tuns-ssh	50	0.50	1 av 1 kjente (0 falske, 0 mangler)	63.14s
tuns-web	50	0.50	0 av 1 kjente (0 falske, 1 mangler)	
iodine-ssh	200	0.50	1 av 1 kjente (0 falske, 0 mangler)	15.67s
iodine-web	200	0.50	1 av 1 kjente (0 falske, 0 mangler)	18.89s
nstx-ssh-mtu1500	200	0.50	0 av 1 kjente (0 falske, 1 mangler)	
nstx-ssh-mtu500	200	0.50	0 av 1 kjente (0 falske, 1 mangler)	
nstx-web	200	0.50	1 av 1 kjente (0 falske, 0 mangler)	3.06s
tuns-ssh	200	0.50	0 av 1 kjente (0 falske, 1 mangler)	
tuns-web	200	0.50	0 av 1 kjente (0 falske, 1 mangler)	

Tabell 5.8: Varierende numpackets på spesielle datasett Kolmogorovkompleksitet

Fordelen med denne teknikken er at den ikke er avhengig av mange klienter for å skape statistikk. Det er en egenskap av en enkelt klient, og veldig tydelig. På grunn av dette bør den også kunne fungere i tilfeller der man har få klienter, slik som på en hotspot-løsning på en kafé eller en lite trafikkert flyplass.

5.5.1 Første forsøk

Denne metoden ble prøvd ut etter metoden for båndbredde. Kravene som ble satt var:

- Mottakerdomenet til minst 98% av spørringene i samplet skal ha vært det samme.

Datasett	num-packets	kolmo-minimum	Positiver	Resp-tid
2010-04-08	20	0.50	1332 av 0 kjente (1332 falske, 0 mangler)	
2010-04-09	20	0.50	1114 av 0 kjente (1114 falske, 0 mangler)	
2010-04-13	20	0.50	1626 av 0 kjente (1626 falske, 0 mangler)	
2010-04-27	20	0.50	1517 av 1 kjente (1516 falske, 0 mangler)	7.29s
2010-04-08	50	0.50	970 av 0 kjente (970 falske, 0 mangler)	
2010-04-09	50	0.50	789 av 0 kjente (789 falske, 0 mangler)	
2010-04-13	50	0.50	1144 av 0 kjente (1144 falske, 0 mangler)	
2010-04-27	50	0.50	1098 av 1 kjente (1097 falske, 0 mangler)	21.32s
2010-04-08	200	0.50	855 av 0 kjente (855 falske, 0 mangler)	
2010-04-09	200	0.50	685 av 0 kjente (685 falske, 0 mangler)	
2010-04-13	200	0.50	1006 av 0 kjente (1006 falske, 0 mangler)	
2010-04-27	200	0.50	974 av 1 kjente (973 falske, 0 mangler)	88.27s
2010-04-27	2000	0.50	956 av 1 kjente (955 falske, 0 mangler)	902.34s
2010-04-27	4000	0.50	956 av 1 kjente (955 falske, 0 mangler)	1310.27s

Tabell 5.9: Resultater generelle datasett Kolmogorovkompleksitet

- Klienter med mindre enn 10 spørringer per sekund ble filtrert vekk.
- Båndbreddeforbruket må minst ha vært 250 byte per sekund i samplet.

Disse filterverdiene ble satt med prøving og feiling. Det mest overraskende var at prosentfaktoren måtte såpass høyt for å fjerne falske positiver. En lengre sample-tidsrom kunne sannsynligvis også påvirket dette i positiv forstand.

Datasett	Lengde [s]	Alarm	Positiver (falske og totalt)
2010-04-08	20	3	1 av 1
2010-04-09	20	3	1 av 1
2010-04-13	20	3	2 av 2
2010-04-27	20	3	1 av 2

Tabell 5.10: Resultater første forsøk mottakerdomene

Resultatene er oppsummert i tabell 5.10. Det er verdt å merke seg at det er to klienter som opptrer som falske positiver av rundt 3000 totalt i hvert datasett. Sett fra synpunktet til en driftsansvarlig vil hvitelisting vil kunne håndtere disse.

I samplet fra 27. april blir klienten som er kjent som IP-over-DNS -klient merket. Oppførselen til den falske positiven i dette tilfellet er at den blir detektert i en kort periode. Resultatene er oppsummert i tabell 5.10. Datasettet er tatt fra klokken 13:30 til 14:30. Den korrekt identifiserte IP-over-DNS -klienten er markert kontinuerlig i alle intervallene i datasettet. Den falske positive opptrer fra klokken 14:26:42 til 14:27:24. Et lengre sampleintervall ville derfor filtrert vekk denne, og beholdt Iodine-klienten.

5.5.2 Andre forsøk

Ettersom denne teknikken gav ganske gode resultater i utgangspunktet, ble ulike verdier for antall samples med alarmer samt sampletid testet. Resultatene vises i tabell 5.11.

Datasett	Lengde [s]	Alarmnivå	Positiver	av disse falske
2010-04-09	5	3	1	1
2010-04-08	5	3	1	1
2010-04-13	5	3	2	2
2010-04-27	5	3	3	2
2010-04-09	30	3	1	1
2010-04-08	30	3	1	1
2010-04-13	30	3	1	1
2010-04-27	30	3	1	0
2010-04-09	5	5	1	1
2010-04-08	5	5	1	1
2010-04-13	5	5	1	1
2010-04-27	5	5	1	0
2010-04-09	5	8	0	0
2010-04-08	5	8	0	0
2010-04-13	5	8	0	0
2010-04-27	5	8	**0 (!)**	0

Tabell 5.11: Resultater andre forsøk mottakerdomene

Oppsummert ser man ut til å få gode resultater ved 5 sekunder sampletid og krav om å bli merket i 5 etterfølgende samples. Reaksjonstiden blir følgelig 25 sekunder. Ved å kreve 8 etterfølgende samples med alarm

per klient ser man at man får en falsk negativ; den kjente IP-over-DNS -klienten blir ikke markert i det hele tatt i løpet av datasettet.

Kolonnen merket "positiver" beskriver at klienten en eller annen gang i løpet av datasettet har blitt merket som IP-over-DNS -klient. En usikkerhet som ikke kommer frem er hvorvidt klienten merkes med en gang, eller om den ligger skjult en stund før aktiviteten blir høy nok til å kunne merkes. En måte å vise dette på kan være å summere hvor lang tid den har vært merket som aktiv, og hvor langt ut i datasettet den først inntraff. Dette er ikke videre utprøvd.

5.5.3 Tredje forsøk

I tredje forsøk ble hendelsesbasert punktprøving brukt i stedet for uniform. En klient sjekkes etter IP-over-DNS -trafikk når den har fått n svar (ett "sample"), uavhengig av hvor lang tid det tar. Deteksjonsgrensen er at minst 98% av spørringene i settet skal ha gått mot samme domene. Pseudokoden for metoden vises i figur 5.2.

Figur 5.2: Pseudokode for tredje forsøk mottakerdomene

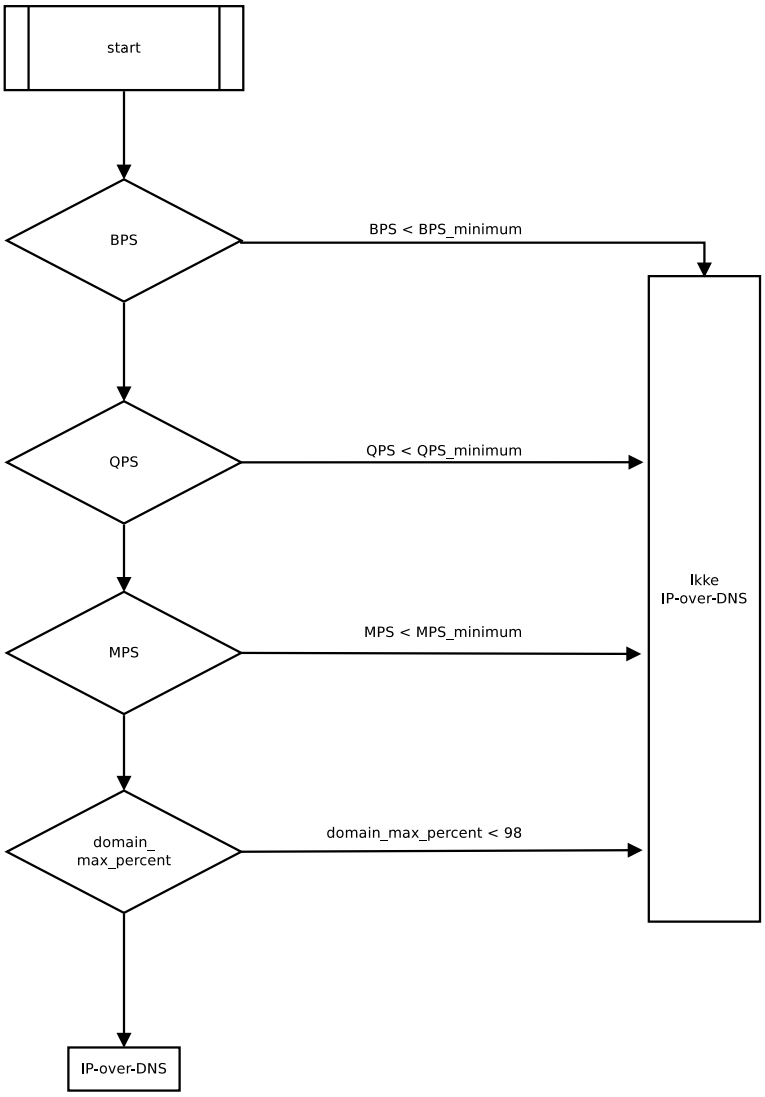
```
def beregn( liste_med_en_klients_pakker ):
    beregn QPS, BPS, MPS basert paa samtlige av klientens pakker.
    for pakke in liste_med_klients_pakker:
        domene = (inntil) tre siste labels i Question-delen \
            av pakken. ``www.item.ntnu.no -> item.ntnu.no``
        buf[klient][domene] += len(pakke)

    sorter buf
    for hver klient:
         hent ut det domenet klienten har sendt mest data til.
         domain_max_percent = domain_max_bytes /
             (totale antall byte overfoert av klient i samplet)

    om BPS < oppgitt grense: return False
    om QPS < oppgitt grense: return False
    om MPS < oppgitt grense: return False

    om domain_max_percent > oppgitt_nivaa:
        return True
    return False
```

Disse verdiene er i utgangspunkt satt som de empiriske minimumsverdiene observert i forarbeidet. Dette er oppsummert i tabell 5.12.
Beslutningsflyten for hvorvidt et sample skal merkes som IP-over-DNS er beskrevet i figur 5.2. Testen er negativt rettet, slik at om en av kriteriene ikke er oppfylt så sies klienten å ikke drive med IP-over-DNS i dette samplet.



Figur 5.2: Beslutningsflyt

Variabel	Verdi
numpackets	30
domain_max_percent	98%
movingaveragedepth	1
bps_minimum	560
qps_minimum	2.27
mps_minimum	240

Tabell 5.12: Deteksjonsgrenser for SSH

Resultat

Med de empiriske verdiene for SSH som konfigurasjon er resultatet at på de kjente datasettene får man **ingen falske positiver**. Kjent IP-over-DNS -klient blir vellykket rapportert.

5.5.4 Båndbredde per domene oppsummert

Dette er den teknikken som fungerer best av de som har blitt testet, og velges som teknikken som brukes til å forsøke å unngå deteksjon.

5.6 Forsøk på å unngå deteksjon

Når en deteksjonsløsning nå er på plass og kan testes i bruk, er neste steg å forsøke å modifisere deler av teknikken til IP-over-DNS slik at de ikke blir detektert.

Noen metoder for båndbreddebegrensning ble prøvd ut uten suksess:

- Bruke linux sin netfilter til å båndbreddebegrense tunnellen. Over tid virker dette, men nøyaktigheten på begrensninger av "bursts" er for lav. Opplevde resultater tyder på at for mye data tillates å sendes før begrensningene trår til og forkaster pakkene. Alarmen går med en gang en TCP-sesjon settes opp.
- Bruke user-space programmet "trickle" til å begrense båndbredde-bruk. Selv på laveste nivå på 1KB/s går alarmen.

Bakgrunnen til å forsøke å båndbreddebegrense var å komme under de empiriske nivåene observert. Om man har en svært streng båndbreddebe-grensning, eller justerer Iodine/TUNS/NSTX til å ikke sende oftere enn n

pakker per sekund, bør man kunne få til dette mot høyere latens og mindre båndbredde i tunnellen.

5.6.1 Små fragmenter

Etter å ha testet TUNS og observert at de små pollepakkene dens ikke gir alarm, virker det som at nivået for gjennomsnittlig pakkestørrelse (satt til 240 byte) er høyt. Om man konfigurerer Iodine til å bruke bruke maksimal fragmentstørrelse (mengde data per DNS-spørring) på 100 byte, blir effektiv gjennomsnittlig pakkestørrelse rundt 180 byte. **Dette fører til at alarmen ikke går, til tross for at tunnellen belastes fullt ut.** Latensen på tunnellen med denne lave fragmentstørrelsen forblir lav, 2-3ms gjennom dnscache på lokalnett, og målinger med iperf over tunnellen oppgir en effektiv båndbredde på 678Kbit/s. (86KiB/s)

Prøving og feiling tilsier at minste fragmentstørrelse nivå Iodine ser ut til å virke er 49 byte. Effektivt ligger pakkene 90 byte over fragmentstørrelsen, som gir rundt 140 byte.

For å hindre at dette skal kunne brukes til å omgå deteksjon, ble nivået for minimum gjennomsnittlig pakkestørrelse justert. I utprøvingen med 190 byte (tilfeldig valgt lavere enn 240) som mps_minimum får man flere innslag av falske positiver i de generelle pakkedumpene fra NTNU. Totalt er det snakk om inntil 5 unike IP-adresser, der nivåene for MPS i alarmene på rundt 170 byte.

Oppsummert er dette en svakhet i deteksjonsmekanismen. Reglene sier at om en av QPS/BPS/MPS er lavere enn oppgitt nivå, så er dette ikke IP-over-DNS. Forsøk på å endre reglene slik at at minst to av disse må være lavere enn konfigurerte verdier gav store mengder falske positiver, og ble forkastet.

MPS må settes til 140 byte for å kunne fange opp disse små fragmentene. Kjøring på generelle datasett gir mange falske positiver (50+). Generelt er oppførselen til IP-over-DNS -klienter mer seiglivet enn vanlige klienter, så numpackets økes for å se om dette hjelper. numpackets=100, 50 og til slutt 70 ble testet. 70 gav ingen falske positiver, og en fornuftig deteksjonstid. Første alarm på kjent IP-over-DNS-klient skjer nå 30 sekunder inn i datasettet, mot 13 sekunder på numpackets=30. De oppdaterte konfigurasjonsverdiene vises i tabell 5.13.

Dette virker som en fungerende balanse mellom deteksjon av faktiske IP-over-DNS -klienter og å unngå å få falske positiver.

Variabel	Verdi
numpackets	70
domain_max_percent	98%
movingaveragedepth	1
bps_minimum	560
qps_minimum	2.27
mps_minimum	140

Tabell 5.13: Oppdaterte deteksjonsgrenser med lavere MPS

5.6.2 Etterkontroll

Programkoden for sanntidsdeteksjon ble lyttestasjonen i laboratoriet, og konfigurasjon satt slik nevnt i tabell 5.13. De tre ulike tunnelleringsteknikkene ble testet manuelt, og resultatene er oppsummert i tabell 5.14.

Andre ideer for å unngå deteksjon

Andre ideer som bør kunne påvirke deteksjon, men som ikke er testet:

- Sende en del DNS-spørringer mot rekurserende navnetjener sånn at andelen pakker til samme domene kommer under 98%.
- Modifisere Iodine/TUNS til å kunne polle med ujevn takt og kan bruke flere enn ett domene.

Listen over slike alternative metoder kan sannsynligvis gjøres veldig lang, og utprøving av alle disse er heller ikke fokuset for dette arbeidet.

5.7 Anbefalt deteksjonsmekanisme

Den mest hensiktsmessige metoden for å detektere IP-over-DNS på nåværende tidspunkt er å måle båndbreddeforbruk per mottakerdomene, slik beskrevet i delkapittel 5.5. Hendelsesbasert punktprøving må brukes for å få god nok nøyaktighet.

Tunneltype	Resultat	Kommentarer
iodine 0.6.0rc1	OK	Oppsetting av tunnel gir ingen alarm. Pinging gjennom tunnellen gir ingen alarm. SSH gjennom tunnellen gir alarm med en gang innlogging er fullført. Ingen kontinuerlig alarm etter innlogging, men hver gang tekst overføres. (f.eks. "ls -ltrs")
TUNS 0.9.2	OK	Latensen over tunnelen er 11ms og en enkel kjøring av iperf gir minimale 11 Kbit/s tilgjengelig båndbredde. SSH fungerer, men kommer i rykk og napp. Alarmen går ved pinging over tunnellen, ssh og når iperf kjøres. Ingen alarm på ren keepalive eller innlogging. Deteksjonsmekanismen fungerer som forventet.
NSTX	OK	1.9ms latens over tunnelen med ping. Ingen alarm ved pinging. iperf sier 73Kbit/s. SSH fungerer, men oppleves som "hakkete". Alarm etter noen terminaldybder med aktivitet. Deteksjonsmekanismen fungerer som forventet.

Tabell 5.14: Resultater manuell etterkontroll

5.8 Avsluttende kommentarer

Programkoden ser ut til å fungere, og konfigurasjonsvariablene vist i tabell 5.13 ser ut til å være de mest effektive. Den gir litt lengre deteksjonstid, opp mot 30 sekunder, men gir ikke falske positiver verken i laboratoriet eller på tilgjengelige datasett.

Kapittel 6

Diskusjon og konklusjon

6.1 Konklusjon

Hypotesen som skulle undersøkes i dette arbeidet var om det var mulig å påvise IP-over-DNS -trafikk gjennom en DNS-server automatisk. **Basert på funnene i kapittel 5 er dette mulig**.

De to teknikkene som var planlagt å bli brukt, en klients trafikk sin Kolmogorov-kompleksitet og mengden med DNS-trafikk en klient sender, viste seg å ikke være de beste metodene. Kompleksiteten til innholdet lot ikke til å entydig være en observator for IP-over-DNS -trafikk. DNS-trafikk per klient hadde for dårlig reaksjonstid, og krever hvitelisting av kjente servere på innsiden.

Teknikken med å se på domenet DNS-trafikken gikk til gav gode resultater. Det er sannsynlig at denne metoden kan benyttes operasjonelt uten å gi så mange falske positiver.

6.2 Originalitet og ytelse

Om man ser bort fra regelbaserte systemer, slik beskrevet i kapittel 4, er dette arbeidet det eneste systemet for deteksjon av IP-over-DNS som er å finne på Internett.

Implementasjonen av teknikkene er gjort tilgjengelig både i vedlegget og på WWW. Systemet har maskinkrav som gjør det gjennomførbart å utføre analysen i sanntid på vanlig maskinvare. En motivert programmerer vil kunne bruke programkoden til å finpusse systemet slik at det kan

brukes kommersielt.

6.3 Fremtidig arbeid

I [CPZB09] diskuterer Chatzis m.fl. hvordan man kan bruke såkalt "Time series analysis" med wavelets på DNS-trafikk for å detektere utbredelsen av epostormer. Dette virker som en spennende teknikk, men matematikken involvert er litt utenfor rekkevidden til forfatteren. Det kunne vært interessant å sett hvordan en slik metode fungerte i forhold til de enkle nivåbaserte metodene beskrevet her.

Det kunne vært interessant å se på rekkefølgen av QTYPE på DNS-forespørsler. Dette har vært brukt i innbruddsdeteksjonssystemer tidligere. En IP-over-DNS -klient bør sende et tydelig annerledes mønster av spørretyper enn en vanlig klient. Dette er noe av det forfatteren av TUNS prøvde å unngå. Å se hvordan det ser ut i praksis kunne vært interessant.

Autokorrelasjon er en metode for å undersøke et signal etter repeterende hendelser. Premisset her er at vanlige DNS-klienter vil spørre en rekke forespørsler på kort tid, og så være stille en stund. IP-over-DNS-klienter vil polle repetitivt, og bør derfor ha klare topper på rundt 1-3 sekunder. En annen variant vil kunne være å kombinere båndbredde per mottakerdomene med denne teknikken, for å fjerne en del bakgrunnsstøy.

Bibliografi

[ASON03] Andre L. M. dos Santos Abhishek Singh Ola Nordström, Chenghuai Lu. Malicious ICMP Tunneling: Defense against the Vulnerability. pages 217–217, 2003.

[CBK09] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):1–58, 2009.

[CPZB09] N. Chatzis, R. Popescu-Zeletin, and N. Brownlee. Email worm detection by wavelet analysis of dns query streams. *IEEE Computational Intelligence Society: CICS 2009, IEEE Symposium on Computational Intelligence in Cyber Security, pp. 53-60*, 2009.

[GV99] Alexander Gammerman and Vladimir Vovk. Kolmogorov Complexity: Sources, Theory and Applications. *The Computer Journal*, 42(4):252–255, 1999.

[Hub10] Bert Hubert. Powerdns recursor 3.2 available. `http://mailman.powerdns.com/pipermail/pdns-announce/2010-March/000127.html`, 2010. [Besøkt 2010-04-21].

[LH07] Robert Lyda and James Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security and Privacy*, 5:40–45, 2007.

[Moc87a] P. Mockapetris. Domain names - concepts and facilities. `http://tools.ietf.org/html/rfc1034`, 1987. [Besøkt 2010-04-19].

[Moc87b] P. Mockapetris. Domain names - implementation and specification. `http://tools.ietf.org/html/rfc1035`, 1987. [Besøkt 2010-04-19].

[NNR09] L. Nussbaum, P. Neyron, and O. Richard. On Robust Covert Channels Inside DNS. *Emerging Challenges for Security, Privacy and Trust. 24th IFIP TC 11 International Information Security Conference, SEC 2009, Proceedings from*, pages 51–62, 2009.

[PF95] Vern Paxson and Sally Floyd. Wide-area traffic: The failure of poisson modeling. *IEEE/ACM Transactions on Networking*, pages 226–244, 1995.

[Sta05] William Stallings. *Cryptography and Network Security (4th Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[TvLL08] Kwan-Wu Chin Tom van Leijenhorst and Darryn Lowe. On the Viability and Performance of DNS Tunneling. *The 5th International Conference on Information Technology and Applications (ICITA 2008)*, 2008.

[Vix99] Paul Vixie. Extension Mechanisms for DNS (EDNS0). `http://tools.ietf.org/html/rfc2671`, 1999. [Besøkt 2010-04-19].

[WP05] A. Wagner and B. Plattner. Entropy based worm and anomaly detection in fast IP networks. In *Enabling Technologies: Infrastructure for Collaborative Enterprise, 2005. 14th IEEE International Workshops on*, pages 172 – 177, june 2005.

Tillegg A

Laboratorieoppsett

For å teste de ulike teknikkene ble det satt opp en testlab med alle påkrevde roller. Et logisk oversiktsbilde er vist i Figur A.1. En liste over involverte maskiner, roller og IP-adresser vises i Tabell A.1.

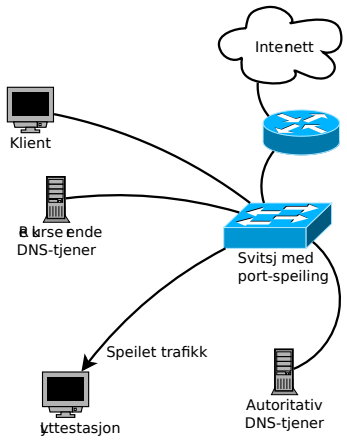
Maskinene involvert kjører Ubuntu Linux 9.10. Dette er standard x86-baserte maskiner, med av varierende fabrikat og alder. Dette kan være en feilkilde i målingene, men med tanke på de lave kravene til ytelse problemet har så bør dette ikke ha særlig innvirkning.

En HP Procurve 1800-8G ethernet-svitsj brukes til å koble sammen maskinene. Denne svitsjen støtter port-speiling. Maskinen *kilo* har to nettverkskort, og all trafikk til og fra porten til *phi* (rekurserende navnetjener) blir speilet ut på denne. Ulik programvare brukes på *kilo* for å analysere innkommende pakkestrøm.

ROLLE	NAVN	IP-ADRESSE	SOFTWARE
Klient	hotel	10.51.1.140	Ubuntu Linux 9.10, iodine, tuns
Lyttestasjon	kilo	10.51.1.100	snort, tcpdump
Rekurserende navnetjener	phi	10.51.1.50	BIND 9.4.2
Autoritativ navnetjener	oscar	10.51.1.150	iodine-server

Tabell A.1: Liste over maskiner i laboratorieoppsett

DNS-sonen "dnslab.nu.mu" ble satt opp. Ettersom laboppsettet befinner seg bak NAT, ble det vanskelig å få en fullstendig delegering på plass. For å løse på dette ble domenet sine navnetjenere hardkodet i konfigurasjonen til rekurserende navnetjener (*kilo*.)



Figur A.1: Oversiktsbilde over laboratorieoppsett

A.1 Oppsett av Iodine

Server

Hentet ned `http://code.kryo.se/iodine/iodine-0.6.0-rc1.tar.gz`. Pakket ut, bygde med make på vanlig vis.

```
root@oscar:~/iodine-0.6.0-rc1/bin# ./iodined -u iodine -t /var/run/iodine \
  -P foo -P foo 10.26.0.1 dnslab.nu.mu

Apr  6 11:38:04 oscar iodined: started, listening on port 53
Apr  6 11:50:57 oscar iodined: started, listening on port 53
```

Klient

Samme første for å bygge som på serveren, bare under tmp/.

```
root@hotel:~/tmp/iodine-0.6.0-rc1/bin# ./iodine 10.51.1.150 dnslab.nu.mu
Enter password:
Opened dns0
Opened UDP socket
Sending DNS queries for dnslab.nu.mu to 10.51.1.150
Autodetecting DNS query type (use -T to override).
Using DNS type NULL queries
```

```
Version ok, both using protocol v 0x00000502. You are user #0
Setting IP of dns0 to 10.26.0.2
Setting MTU of dns0 to 1130
Server tunnel IP is 10.26.0.1
Testing raw UDP data to the server (skip with -r)
Server is at 10.51.1.150, trying raw login: OK
Sending raw traffic directly to 10.51.1.150
Connection setup complete, transmitting data.
Detaching from terminal...
# legge default-routen ut gjennom tunnellen
root@hotel:~/tmp/iodine-0.6.0-rc1/bin# ip route add default via 10.26.0.1
```

Unngå RAW mode

Ettersom iodine-serveren er direkte nåbar for klienten, vil iodine bruke RAW mode. Man kan be Iodine hoppe over testen med å bruke opsjonen "-r", men for å gjøre det hele litt mer virkelighetsnært ble dette implementert med en brannveggregel i stedet.

```
root@oscar:~/iodine-0.6.0-rc1/bin# iptables -A INPUT -s 10.51.1.140 \
  -p udp --dport 53  -j DROP
```

A.2 Oppsett av TUNS

Grunnleggende installasjon

Hentet `http://www.loria.fr/~lnussbau/files/tuns-0.9.2.tgz` som linket til fra `http://www.loria.fr/~lnussbau/tuns.html`.
Det er Ruby, og noen avhengigheter må installeres:

```
root@oscar:~/tuns-0.9.2# apt-get install rake ruby1.8-dev
root@oscar:~/tuns-0.9.2/base32-0.1.1# rake
root@oscar:~/tuns-0.9.2# cp base32-0.1.1/ext/base32.so .
```

Det fullfører installasjonsstegene som er beskrevet i README-filen som ligger med.

Kjøring av server

```
root@oscar:~/tuns-0.9.2# ./tuns-server --debug -d dnslab.nu.mu
```

Klient

Samme steg som på server.
Ser ut til å foretrekke 192.168.53.0/24 på endepunktene av tunnellen. ("linknettet")

```
lkarsten@hotel:~/tmp/tuns-0.9.2$ sudo ./tuns-client \
    -d dnslab.nu.mu -n 10.51.1.50


lkarsten@hotel:~$ sudo ip route add default via 192.168.53.2
root@hotel:~# ip route
192.168.53.2 dev tun0  scope link
10.51.1.0/24 dev eth0  proto kernel  scope link  src 10.51.1.140  metric 1
169.254.0.0/16 dev eth0  scope link  metric 1000
default via 192.168.53.2 dev tun0
root@hotel:~#
root@hotel:~# ping www.ntnu.no -c 5
PING illa4.itea.ntnu.no (129.241.18.154) 56(84) bytes of data.
64 bytes from illa4.itea.ntnu.no (129.241.18.154): \
    icmp_seq=1 ttl=56 time=31.0 ms
64 bytes from illa4.itea.ntnu.no (129.241.18.154): \
    icmp_seq=2 ttl=56 time=32.6 ms
64 bytes from illa4.itea.ntnu.no (129.241.18.154): \
    icmp_seq=3 ttl=56 time=22.3 ms
64 bytes from illa4.itea.ntnu.no (129.241.18.154): \
    icmp_seq=4 ttl=56 time=24.8 ms
64 bytes from illa4.itea.ntnu.no (129.241.18.154): \
    icmp_seq=5 ttl=56 time=23.6 ms

--- illa4.itea.ntnu.no ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4006ms
rtt min/avg/max/mdev = 22.393/26.919/32.659/4.130 ms
root@hotel:~#
```

Ser ut til å virke.

```
root@hotel:~# tracepath www.ntnu.no
 1:  192.168.53.1 (192.168.53.1)                            0.293ms pmtu 140
 1:  192.168.53.2 (192.168.53.2)                         1024.971ms
 1:  192.168.53.2 (192.168.53.2)                         5025.965ms
 2:  no reply
 3:  1.80-202-221.nextgentel.com (80.202.221.1)           25.085ms
 4:  217-13-1-70.dd.nextgentel.com (217.13.1.70)          54.231ms
 5:  trd-gw1.uninett.no (193.156.93.1)                    25.492ms asymm  6
 6:  ntnu-gsw.ntnu.no (158.38.0.222)                      25.679ms
 7:  dragv-gsw.ntnu.no (129.241.76.6)                     23.292ms
 8:  dragv-gsw2.ntnu.no (129.241.76.214)                  30.756ms
 9:  illa4.itea.ntnu.no (129.241.18.154)                  51.677ms reached
```

```
     Resume: pmtu 140 hops 9 back 56
root@hotel:~#
```

Forsøk på å hente websider over gir rapportert (med mtr) rundt 5-10KB/s over tunnelinterfacet. Dette går sakte. Klientmaskinen er merkbart treigere.

A.3 Oppsett av NSTX

Server

Programvaren er god og gammel, og også pakket i Debian sitt programvarearkiv. Installasjon blir derfor enklere. Versjonen i arkivet er 1.1-beta6.

```
root@oscar:~# apt-get install nstx
Modifiserte /etc/default/nstx slik:
"""
NSTX_DOMAIN="dnslab.nu.mu"
start_nstxd=yes
ifup_tun=tun0
"""
root@oscar:/etc/default# ifconfig tun0 10.26.0.1 netmask 255.255.255.0 up
```

Klient

Ubuntu har samme versjon av NSTX som på serveren. 1.1-beta6.

```
lkarsten@hotel:~$ sudo apt-get install nstx
Modifiserte /etc/default/nstx slik:
"""
NSTX_DOMAIN="dnslab.nu.mu"
NSTX_DNS_SERVER=10.51.1.50
start_nstxcd=yes
"""
root@hotel:~# /etc/init.d/nstxcd start
Starting nstxcd: nstxcd.
root@hotel:~# ifconfig tun0 10.26.0.2 netmask 255.255.255.0 up

root@hotel:/etc/default# ifconfig tun0 10.51.26.2 netmask 255.255.255.0 up
root@hotel:~# ping 193.212.1.10 -c 2
PING 193.212.1.10 (193.212.1.10) 56(84) bytes of data.
64 bytes from 193.212.1.10: icmp_seq=1 ttl=120 time=37.7 ms
64 bytes from 193.212.1.10: icmp_seq=2 ttl=120 time=42.8 ms

--- 193.212.1.10 ping statistics ---
```

```
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 37.767/40.301/42.836/2.542 ms
```

Virker fint når man bare kjører ping, men så snart man får mye data henger det litt. HOWTOen nevner MTU-problemer.

```
root@hotel:~# ifconfig tun0 10.26.0.2 netmask 255.255.255.0 up mtu 500
root@oscar:/etc/default# ifconfig tun0 10.26.0.1 \
    netmask 255.255.255.0 up mtu 500
```

A.4 Oppsett for live-testing

For å få gode og sammenlignbare data ble en IP-over-DNS-tunnel basert på Iodine satt opp fra en maskin på lesesalen og ut gjennom NTNU sin offisielle DNS-server: 129.241.0.201.

dnssonen er dnslab2.nu.mu, som er delegert til maskinen yankee.samfundet.no. Iodine-serveren kjøres på yankee.

Inni tunnellen kjøres det en SSH-sesjon. Inni SSH står kommandoen "watch -n1 date", som er tenkt å illustrere interaktiv bruk. Denne vil sende litt trafikk over tunnellen hvert sekund.

```
root@yankee:/usr/src/iodine-0.6.0-rc1/bin# ./iodined  -D -u iodine \
    -t /var/run/iodine -P foo -P foo 10.100.0.1 dnslab2.nu.mu
Debug level 1 enabled, will stay in foreground.
Add more -D switches to set higher debug level.
Opened dns0
Setting IP of dns0 to 10.100.0.1
Setting MTU of dns0 to 1130
Opened UDP socket
Listening to dns for domain dnslab2.nu.mu

IN   login raw, len 16, from user 0
IN   ping raw, from user 0
PING pkt from user 1, ack for downstream 0/0
```

```
root@phi:~/iodine-0.6.0-rc1/bin# ./iodine  -P foo \
    -r 129.241.0.201 dnslab2.nu.mu
Opened dns0
Opened UDP socket
Sending DNS queries for dnslab2.nu.mu to 129.241.0.201
```

```
Autodetecting DNS query type (use -T to override).
Using DNS type NULL queries
Version ok, both using protocol v 0x00000502. You are user #1
Setting IP of dns0 to 10.100.0.3
Setting MTU of dns0 to 1130
Server tunnel IP is 10.100.0.1
Skipping raw mode
Using EDNS0 extension
Switching upstream to codec Base128
Server switched upstream to codec Base128
No alternative downstream codec available, using default (Raw)
Switching to lazy mode for low-latency
Server switched to lazy mode
Autoprobing max downstream fragment size... (skip with -m fragsize)
768 ok.. 1152 ok.. ...1344 not ok.. ...1248 not ok..
     ...1200 not ok.. 1176 ok.. 1188 ok.. will use 1188-2=1186
Setting downstream fragment size to max 1186...
Connection setup complete, transmitting data.
Detaching from terminal...
root@phi:~/iodine-0.6.0-rc1/bin#
```

A.5 Shaping

Hentet ned http://puzzle.dl.sourceforge.net/project/cbqinit/cbqinit/0.7.3/cbq.init-v0.7.3 , innførte små endringer for å lese konfigurasjonsfiler fra en annen path enn /etc/sysconfig/.

```
#WEIGHT=5000Kbit
#PEAK=100Kbit

DEVICE=eth0,1000Mbit,100Mbit
RATE=1kbit
#WEIGHT ~= RATE / 10
WEIGHT=500b
#500b
PEAK=500bit

#       Maximal peak rate for short-term burst traffic. This allows you
#       to control the absolute peak rate the class can send at, because
#       single TBF that allows 256Kbit/s would of course allow rate of
#       512Kbit for half a second or 1Mbit for a quarter of second.

RULE=10.51.0.200
REALM=hotel,enjoy
MARK=10
```

``./cbq.init-v0.7.3 start'' for å slå på.

```
``iperf -s`` på 10.51.0.200.

''bwm-ng -u bits`` for å måle bw.
''root@hotel:~# iperf  -c 10.51.0.200``
```

setter opp iodine, kjører iperf over.

```
root@hotel:~/tmp/iodine-0.6.0-rc1/bin# iperf -t 20 -c 10.26.0.1
------------------------------------------------------------
Client connecting to 10.26.0.1, TCP port 5001
TCP window size: 16.0 KByte (default)
------------------------------------------------------------
[  3] local 10.26.0.2 port 58003 connected with 10.26.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-20.9 sec  3.19 MBytes  1.28 Mbits/sec
```

alarm går så det suser:

```
domain_max_percent alarm: 100.0 more than configured 98
{'bps': 26554.076895953822,
 'bytes': 7305,
 'bytes_kolmo': 7232,
 'domain_max_name': 'dnslab.nu.mu',
 'domain_max_percent': 100.0,
 'first_ts': datetime.datetime(2010, 6, 30, 13, 49, 53, 862624),
 'last_ts': datetime.datetime(2010, 6, 30, 13, 49, 54, 137723),
 'mps': 243.5,
 'mtbq': 0.0094861724137931033,
 'qps': 109.05165049673028,
 'queries': 30,
 'sample_length': 0.27509899999999998}
None
alarm: 10.51.0.140
Probable IP-over-DNS client: 10.51.0.140
    (100\% of 7305 bytes from domain dnslab.nu.mu)
```

Trickle

"apt-get install trickle" for å installere. Forsøker å kopiere over en fil med scp:

```
    lkarsten@hotel:~\$ trickle -v -s -d 1 -u 1 -w 1 -t 1 -l 1 \
        scp  tuns-0.9.2.tgz root@10.26.0.1:deleteme.tgz
```

Samme resultat, alarmen går.

Tillegg B

Kildekode

B.1 Oppbygning

Kildekoden er i to hoveddeler: innlastingskode og gjenkjenningskode. Begrepene "BW", "Kolmo" og "DST" brukes til å indikere de tre hovedtypene deteksjon, båndbredde, Kolmogorov-kompleksitet og data per utgående domene.

Innlastingskoden henter enten pakker fra PCAP-filer, eller i realtime-iodFinder.py sitt tilfelle, direkte fra et nettverkskort i sanntid. pcapsplit-xxx.py indikerer uniform samling, pcapclient-xxx.py indikerer hendelsesbasert sampling.

Gjenkjenningskoden er gruppert i ulike iodFinder*.py -filer. (iod = IP-over-DNS)

B.2 Forhåndsanalyse

Figur B.1: trekk-ut-karakteristikk-datasett.py

```
#!/usr/bin/python
# .- coding: utf-8 -.
#
# Oppsummer et datasett
#

import os, sys
import pcapy
from impacket.ImpactDecoder import EthDecoder, LinuxSLLDecoder, UDPDecoder
from TemporalStorage import TemporalStorage, MaxSamplesReached
from iodFinder import iodFinder

class Decoder():
    def __init__(self, pcapObj):
        datalink = pcapObj.datalink()
        if pcapy.DLT_EN10MB == datalink:
            self.decoder = EthDecoder()
        elif pcapy.DLT_LINUX_SLL == datalink:
            self.decoder = LinuxSLLDecoder()
        else:
            raise Exception("Datalink type not supported: " % datalink)

        self.pcap = pcapObj
        self.connections = {}
        self.UDPDecoder = UDPDecoder()

    def packetHandler(self, hdr, data):
        global tstore, eagle, first_ts
        p = self.decoder.decode(data)

        # hopp over ipv6-pakkene, impacket st0tter det ikke.
        ptype = "%x" % ord(p.get_data_as_string()[0])
        if ptype == "60":
            return

        ip = p.child()
        try:
            # only accept udp packets
            if ip.get_ip_p() != 17:
                return

        except Exception, e:
            print ip
            print >>sys.stderr, e
            raise Exception
            return

        udp = ip.child()
        data = udp.get_data_as_string()

        dns1 = ord(data[2])
        dns2 = ord(data[3])

        dnscode = "%x%x" % (dns1, dns2)

        # standard query response, no error
```

```
        if not dnscode == "8180":
            return
        res = tstore.add( hdr, p )
        # if res returns something else than None, a new sample is ready.
        if res:
            execute( tstore )
    def start(self):
        self.pcap.loop(0, self.packetHandler)


class iodCharacteristics(iodFinder):
    def __init__(self, slen):
        iodFinder.__init__(self, slen)

    def clistore_sum(self):
        r = {}
        for key in self.clistore.values()[0]:
            for client in self.clistore.keys():
                try:
                    r[key] += self.clistore[client][key]
                except KeyError:
                    r[key] = self.clistore[client][key]
        return r

    def report(self):
        clisum = self.clistore_sum()
        sumkeys = [ "bytes","queries" ]

        print "%.0f & %.0f & %.0f B & %.0fs \\\\ \\hline " % \
            (len(self.clistore.keys()),
             clisum["queries"],
             clisum["bytes"],
             self.sample_length,
             )
        print

def execute( tstore ):
    global eagle
    dataset_length = tstore.range()[1] - tstore.range()[0]
    eagle.sample_length = dataset_length
    eagle.add( tstore.get_data())
    eagle.preprocess()
    eagle.report()
    eagle.reset()
    tstore.expunge()

if __name__ == "__main__":
    filename = sys.argv[1]
    p = pcapy.open_offline(filename)

    import os.path
    print "%s & " % os.path.basename(filename),

    sample_length = 99999999999
    tstore = TemporalStorage(period= sample_length)
    eagle = iodCharacteristics( sample_length )

    first_ts = None

    try:
        Decoder(p).start()
        # siste rest
```

```
        execute( tstore )

    except MaxSamplesReached:
        pass
    except KeyboardInterrupt:
        pass

}}
```

B.3 Båndbredde

Figur B.2: pcapsplit-bw.py

```
#!/usr/bin/python
# .- coding: utf-8 -.
#
# Program for aa lese PCAP-filer og kjoere dem gjennom
# iodFinderBW.
#
import os, sys
import pcapy
from impacket.ImpactDecoder import EthDecoder
from impacket.ImpactDecoder import LinuxSLLDecoder, UDPDecoder
from TemporalStorage import TemporalStorage, MaxSamplesReached
from iodFinderBW import iodFinderBW

class Decoder():
    def __init__(self, pcapObj):
        datalink = pcapObj.datalink()
        if pcapy.DLT_EN10MB == datalink:
            self.decoder = EthDecoder()
        elif pcapy.DLT_LINUX_SLL == datalink:
            self.decoder = LinuxSLLDecoder()
        else:
            raise Exception("Datalink type not supported: " % \
                datalink)

        self.pcap = pcapObj
        self.connections = {}
        self.UDPDecoder = UDPDecoder()

    def packetHandler(self, hdr, data):
        global tstore, eagle, first_ts
        p = self.decoder.decode(data)

        # hopp over ipv6-pakkene, impacket st0tter det ikke.
        ptype = "%x" % ord(p.get_data_as_string()[0])
        if ptype == "60":
            return

        ip = p.child()
        try:
            # only accept udp packets
            if ip.get_ip_p() != 17:
                return

        except Exception, e:
            print ip
            print >>sys.stderr, e
            raise Exception
            return

        udp = ip.child()
        data = udp.get_data_as_string()

        dns1 = ord(data[2])
        dns2 = ord(data[3])

        dnscode = "%x%x" % (dns1, dns2)
```

```
        # standard query response, no error
        if not dnscode == "8180":
            return

        res = tstore.add( hdr, p )

        # if res returns something else than None, a new sample
        # is ready.
        if res:
            eagle.add( tstore.get_data() )
            eagle.preprocess()
            eagle.report()

            eagle.reset()
            tstore.expunge()


    def start(self):
        self.pcap.loop(0, self.packetHandler)

if __name__ == "__main__":
    filename = sys.argv[1]
    p = pcapy.open_offline(filename)

    sample_length = 10
    tstore = TemporalStorage(period=sample_length)
    eagle = iodFinderBW( sample_length )

    first_ts = None
    try:
        Decoder(p).start()
    except MaxSamplesReached:
        pass
    except KeyboardInterrupt:
        pass

#    print dir(eagle) # .banditstore

}}
```

Figur B.3: iodFinder.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# Generell iodFinder-klasse, subklasses av
# de andre mer spesifikke.

class iodFinder():
    """
    Methods to find IP-over-DNS traffic in a set of Impacket data
    packets
    """
    import zlib
    from datetime import datetime
    def __init__(self, slen):
        self.reset()
        # iodFinder is used to analyze samples of n seconds worth
        # of packets. Adjust qps/bps values by this to get proper
        # bytes per second.
        self.sample_length = float(slen)
```

```
        pass

    def reset(self):
        # ts of first data point in sample.
        self.sample_start_ts = None
        self.clistore = {}

    def add(self, packetlist):
        for hdr, p in packetlist:
            ip = p.child()
            udp = ip.child()
            data = udp.get_data_as_string()
            ipaddr = ip.get_ip_dst()

            #print "adding ipaddr %s" % ipaddr

            labels = self._querylabels(data[12:])
            domainkey = ".".join(labels[-3:])

            if not self.clistore.get(ipaddr):
                self.clistore[ipaddr] = { 'queries': 0,
                            'bytes': 0,
                            'bytes_kolmo': 0,
                            # mean time between queries
                            'mtbq': 0.0,
                            'mtbq_store': [],
                            'domain_store': {},
                            }

            #print len(data),
            data_compressed = self.zlib.compress(data)

            self.clistore[ipaddr]["bytes_kolmo"] += len(data_compressed)
            self.clistore[ipaddr]["bytes"] += len(data)
            self.clistore[ipaddr]["queries"] += 1

            ts = hdr.getts()[0]

            if not self.sample_start_ts:
                #print "setting first sample ts to ", ts
                self.sample_start_ts = \
                    self.datetime.fromtimestamp(ts)


            self.clistore[ipaddr]["mtbq_store"].append( ts )

            try:
                self.clistore[ipaddr]["domain_store"][domainkey] += len(data)
            except KeyError:
                self.clistore[ipaddr]["domain_store"][domainkey] = len(data)


    def preprocess(self):
        for key, val in self.clistore.items():
            val["mtbq_store"].sort()
            val["mtbq_deltas"] = []

            # n^2 er min venn.
            for i in range(1, len( val["mtbq_store"])):
                delta = val["mtbq_store"][i] - val["mtbq_store"][i-1]
                val["mtbq_deltas"].append( delta )
```

```
            delta_sum = sum( val["mtbq_deltas"] )
            numdeltas = len( val["mtbq_deltas"] )
            if numdeltas == 0:
               val["mtbq"] == None
            else:
                val["mtbq"] = float(delta_sum) / numdeltas

            # mean packet size
            val["mps"] = (float(val["bytes"]) / val["queries"] )

            del val["mtbq_store"]
            del val["mtbq_deltas"]

            val["domain_max_bytes"] = 0
            val["domain_max_name"] = None

            for domain, domainbytes in val["domain_store"].items():
                if domainbytes > val["domain_max_bytes"]:
                    val["domain_max_bytes"] = domainbytes
                    val["domain_max_name"] = domain

            val["domain_max_percent"] = \
                (100 * float(val["domain_max_bytes"]) / \
                    val["bytes"])

            val["qps"]=float(val["queries"]) /self.sample_length
            val["bps"]=float(val["bytes"])   /self.sample_length

            del val["domain_max_bytes"]
            del val["domain_store"]

    def report(self, header=True):
        raise Exception, "SUBCLASS ME, PLEASE!"

    def stats_formatted(self):
        r = self.stats()
        print "%s\t" % self.sample_start_ts,
        print "average\t",
        if False:
            print "%.2f +-%.2f\t" % (r["mtbq"][0], r["mtbq"][1]),
            print "\t\t\t",
            print "%.2f +-%.2f\t" % (r["bytes"][0], r["bytes"][1]),
            print "%.2f +-%.2f\t" % (r["mps"][0], r["mps"][1]),
            print "%.2f +-%.2f\t" % (r["queries"][0], r["queries"][1]),
        else:

            print "%.2f\t" % (r["mtbq"][0]),
            print "\t\t\t",
            print "%.2f\t" % (r["bytes"][0]),
            print "%.2f\t" % (r["mps"][0]),
            print "%.2f\t" % (r["queries"][0]),

        print


    def stats2_header(self):
        """
        Gir en utskrivbar header for stats2() sitt
        innhold.

        Lite brukt.
        """
```

```
        r = ["ts"]

        for i in ["mtbq","mps","bps"]:
            for j in ["iodine","mean", "stdev"]:
                r.append( "%s_%s" % (i, j))

        r += ["numclients"]
        r += ["qps_per_cli"]
        return r

    def stats2(self):
        """
        Henter ut en tekstvariant av statistikken
        til et sample.

        Lite brukt.

        keys:
        tid unike_klienter iodine_bytes iodine_x i_y, avg_foo
        avg_bar, avg_qux
        """

        r = self.stats()
        iod = self.clistore.get("129.241.208.220") or {}

        res = [ "%s" % self.sample_start_ts ]

        for i in ["mtbq","mps","bps"]:
            iodine_value = iod.get(i)
            if not iodine_value:
                iodine_value = "NA"
            else:
                iodine_value = "%.2f" % float(iodine_value)
            res.append( "%s" % iodine_value)
            res.append( "%.2f" % r.get(i)[0] )
            res.append( "%.2f" % r.get(i)[1] )

        # numclients og qps_per_cli
        res += [ str(len(self.clistore)) ]
        qps_per_cli=r.get("qps")[0] / float(len(self.clistore))
        res += [ "%.6f" % (qps_per_cli) ]

        return res

    def stats(self):
        """
        Find sample statistics

        mean
        stdev

        of:
            mean packet size.
            mean time between queries.
            numqueries
            numbytes

        """
        from statlib import stats
        keys = ["mps","mtbq","qps","bps"]
        res = {}
        for name in keys:
```

```
            # dra ut data. litt hummer&kanari om det er string
            # eller ei, saa cast alt til float.
            vector = [ float(i[name]) for i in self.clistore.values() ]

            if len(vector) == 0:
                raise Exception, "No values found for key %s" % \
                    name

            elif len(vector) == 1:
                res[name] = ( vector[0], 0.0 )
            else:
                res[name] = (
                    stats.mean( vector ),
                    stats.stdev( vector)
                )
        return res


    def _querylabels(self, data):
        """
        Trekk ut en liste av labels i Question-delen av et
        DNS-svar.
        """
        i = 0
        labels = []
        while True:
            # print querydata[i]
            llen = int("%02x" % ord(data[i]), 16)
            #print type(llen), llen

            if llen == 0:
                break

            s1 = i+1
            s2 = s1 + llen
            label = data[s1:s2]

            labels.append(label)
            i = i + llen +1

        return labels


}}
```

Figur B.4: iodFinderBW.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# Kode for aa analysere uniformt samplede datasett
# for klienter som sender statistisk svaert mye data.
#

from iodFinder import iodFinder

class iodFinderBW(iodFinder):
    def __init__(self, slen):
        iodFinder.__init__(self, slen)
        # three last lists of suspects.
        self.persistant_store = [ [], [], [] ]
```

```
        self.tick = 0

    def analyze(self):
        """
        Analyze the current sample for high bandwidth
        clients
        """
        r = self.stats()
        smean = r["bps"][0]
        sstdev = r["bps"][1]
        outlier_limit = smean + 3 * sstdev

        print "Sample average:\t %.4f bps" % smean
        print "Sample stdev:\t%.4f " % sstdev
        print "Outlier limit:\t%.4f bps" % outlier_limit

        suspects = []
        # for hver ip observert
        for ip in self.clistore.keys():
            if self.clistore[ip]["bps"] < outlier_limit:
                continue
            suspects.append(ip)
        return {'suspects': set(suspects),
                'mean': smean, 'stdev': sstdev}

    def timely_analysis(self):
        self.tick = self.tick + 1
        self.persistant_store[2] = \
            self.persistant_store[1]
        self.persistant_store[1] = \
            self.persistant_store[0]
        self.persistant_store[0] = \
            self.analyze()["suspects"]

        ps = self.persistant_store
        bandits = \
        ps[0].intersection(ps[1]).intersection(ps[2])
        return bandits

    def report(self, header=True):
        # foerste variant uten snitting over tid:
        bandits = self.analyze()["suspects"]

        # andre variant:
        #bandits = self.timely_analysis()

        if len(bandits) == 0:
            #print "nothing found"
            return
        else:
            for ip in bandits:
                strings = ["ip","domain_max_name"]
                print "Probable IP-over-DNS client: "
                print "%s (%.f bytes received in sample)" % \
                    ( ip,
                    self.clistore[ip]["bytes"],
                    ),
                print

}}
```

B.4 Kolmogorovkompleksitet

Figur B.5: pcapclient-kolmo.py

```
#!/usr/bin/python
# .- coding: utf-8 -.
#
# wrapper for aa lese inn pcapfiler og
# kjoere dem gjennom iodFinderKOLMO.
#
# NB: Denne haandterer n pakker klient, i
# stedet for aa sample hver n-te sekund.
#

import os, sys
import pcapy
from impacket.ImpactDecoder import EthDecoder
from impacket.ImpactDecoder import LinuxSLLDecoder, UDPDecoder

from ClientStorage import ClientStorage, PacketClientStorage
from ClientStorage import ByteClientStorage
from iodClientFinder import iodClientFinder
from iodClientFinderKolmo import iodClientFinderKolmo

class Timeout(Exception): pass

class Decoder():
    import datetime
    def __init__(self, pcapObj):
        datalink = pcapObj.datalink()
        if pcapy.DLT_EN10MB == datalink:
            self.decoder = EthDecoder()
        elif pcapy.DLT_LINUX_SLL == datalink:
            self.decoder = LinuxSLLDecoder()
        else:
            raise Exception("Datalink type not supported: " % \
                datalink)

        self.pcap = pcapObj
        self.UDPDecoder = UDPDecoder()

    def packetHandler(self, hdr, data):
        global clistore, eagle, max_runtime, veryfirstpacket
        p = self.decoder.decode(data)

        #packet_ts = hdr.getts()[0]
        tstuple = hdr.getts()
        ts = self.datetime.datetime.fromtimestamp(tstuple[0])
        packet_ts = ts.replace(microsecond=tstuple[1])

        if not veryfirstpacket:
            veryfirstpacket = packet_ts

        stoptime = veryfirstpacket + self.datetime.timedelta(seconds=max_runtime)
        if packet_ts >= stoptime:
            raise(Timeout, '%s seconds reached' % max_runtime)

        # hopp over ipv6-pakkene, impacket st0tter det ikke.
        ptype = "%x" % ord(p.get_data_as_string()[0])
        if ptype == "60":
            return
```

```
        ip = p.child()
        try:
            # only accept udp packets
            if ip.get_ip_p() != 17:
                return

        except Exception, e:
            print ip
            print >>sys.stderr, e
            raise Exception
            return

        udp = ip.child()
        data = udp.get_data_as_string()

        dns1 = ord(data[2])
        dns2 = ord(data[3])

        dnscode = "%x%x" % (dns1, dns2)

        # standard query response, no error
        if not dnscode == "8180":
            return

        client = ip.get_ip_dst()
        res = clistore.add( client, hdr, p )

        # if res returns something else than None, a new sample is ready.
        if res:
            #print "yield %s packets or bytes for client %s" % (res, client)
            eagle.add( clistore.get_data(client) )
            eagle.preprocess(client)
            eagle.report(client)

            eagle.reset(client)
            clistore.expunge(client)


    def start(self):
        self.pcap.loop(0, self.packetHandler)

if __name__ == "__main__":
    filename = sys.argv[1]
    p = pcapy.open_offline(filename)

    # stopp etter aa ha parset pakker for n sekunder fra fila.
    #max_runtime = 300
    max_runtime = 999999999

    config = {
        'dev': 'eth1',
        #'kolmo_minimum': 0.50,
        'kolmo_minimum': 0.80,
        'numpackets': 20,
        #'numpackets': 100,
        'samplebytes': 10000,
        'movingaveragedepth': 1,
    }

    clistore = PacketClientStorage(packets=config["numpackets"])
    #clistore = ByteClientStorage(bytes=config["samplebytes"])
```

```
    eagle = iodClientFinderKolmo( config  )

    veryfirstpacket = None

    try:
        Decoder(p).start()
        print "dekoder returnerte"
        # faa med seg det siste ~halve samplet.
    except KeyboardInterrupt:
        pass
    except Timeout:
        pass

    for client in clistore.get_clients():
        clidata = clistore.get_data(client)
        if len(clidata) == 0:
            continue
        eagle.add( clidata )
        eagle.preprocess(client)
        eagle.report(client)

    from EagleReporter import EagleReporter, EagleLatexReporter
    reporter = EagleLatexReporter( config, filename, eagle.banditstore, \
        veryfirstts=veryfirstpacket )
    reporter.report()


}}
```

Figur B.6: iodClientFinderKolmo.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
#
# Kolmogorov!

from iodClientFinder import iodClientFinder

class iodClientFinderKolmo(iodClientFinder):
    def __init__(self, config):
        iodClientFinder.__init__(self, config)
        self.persistant_store = {}
        self.tick = {}

        self.banditstore = {}
        self.sample_stop_ts = None

    def analyze(self, client):
        """
        Gir denne klienten noen advarsel/alarm i dette samplet?
        returnerer True, false.

        Analyze the current sample """
        verbose = True
        veryverbose = True

        data = self.clistore[client]

        #print data["kolmoratio"]
        #print data["kolmoratio"]
        #return False
```

```
        #if client == "129.241.208.220":
        #    print data

        import pprint
        if data["kolmoratio"] > self.config["kolmo_minimum"]:
           #pprint.pprint(data)
           return True
        return False

    def timely_analysis(self, client, depth, verbose=False):
        """
        Analyse per klient, over tid. Om en klient sitt datasett
        gir alarm, vil det lagret her.  Maalet er aa se etter
        etterfoelgende alarmer, saa det vil fungere aa gruppere per
        bruker
        """
        # shift
        if not self.tick.has_key(client):
            self.tick[client] = 0
        self.tick[client] = self.tick[client] + 1
        #for i in range(7, 2):
        #if depth > 8:
        #    raise Exception, "not supported"

        if not self.persistant_store.has_key(client):
            # n last lists of suspects.
            self.persistant_store[client] = \
                [None]* depth

        ps = self.persistant_store
        c = client

        if depth == 1:
            return self.analyze(c)

        if verbose:
            print "pretick %-20s %s: %s" % \
               (client, self.tick[client], ps[client])
        # shift right
        # hva om den er to?
        if depth >= 2:
            for i in range(depth-1, 0, -1):
                #print "shift %s" % i
                self.persistant_store[c][i] = self.persistant_store[c][i-1]

        self.persistant_store[c][0] = self.analyze(c)

        if verbose:
            print "   tick %-20s %s: %s" % \
               (client, self.tick[client], ps[client])

        # AND sammen listen over true/false paa de siste n
        # samples.
        warnings = ps[c][0]
        for i in range(0, depth):
            warnings = warnings and ps[c][i]
        #print "final: ", warnings
        return warnings # alarm

    def pprint(self, d):
        import pprint
```

```
        pprint.pprint(d)

    def report(self, client, header=True):
        # lite data, aldri alarm.
        if self.clistore[client]["queries"] < 2:
            return None

        alarm = self.timely_analysis(client, \
            self.config["movingaveragedepth"], verbose=True)
        if not alarm:
            return None

        if not self.banditstore.has_key(client):
            self.banditstore[client] = []
        self.banditstore[client] += [ self.clistore[client] ]

        r = "Probable IP-over-DNS client:" + \
            "%s (%.0f%% of %.0f bytes from domain %s)" % \
                (
                client,
                self.clistore[client]["domain_max_percent"],
                self.clistore[client]["bytes"],
                self.clistore[client]["domain_max_name"],
                ),
        return " ".join(r)


}}
```

B.5 Per utgående domene

Figur B.7: pcapsplit-dst.py

```
#!/usr/bin/python
# .- coding: utf-8 -.
# wrapper for aa lese inn pcapfiler og kjoere dem gjennom iodFinderDST.
#

import os, sys
import pcapy
from impacket.ImpactDecoder import EthDecoder, LinuxSLLDecoder, UDPDecoder

from TemporalStorage import TemporalStorage, MaxSamplesReached
#from iodFinder import iodFinder
from iodFinderDST import iodFinderDST

class Decoder():
    def __init__(self, pcapObj):
        datalink = pcapObj.datalink()
        if pcapy.DLT_EN10MB == datalink:
            self.decoder = EthDecoder()
        elif pcapy.DLT_LINUX_SLL == datalink:
            self.decoder = LinuxSLLDecoder()
        else:
            raise Exception("Datalink type not supported: " % datalink)

        self.pcap = pcapObj
        self.connections = {}
        self.UDPDecoder = UDPDecoder()

    def packetHandler(self, hdr, data):
        global tstore, eagle, first_ts
        p = self.decoder.decode(data)

        # hopp over ipv6-pakkene, impacket st0tter det ikke.
        ptype = "%x" % ord(p.get_data_as_string()[0])
        if ptype == "60":
            return

        ip = p.child()
        try:
            # only accept udp packets
            if ip.get_ip_p() != 17:
                return

        except Exception, e:
            print ip
            print >>sys.stderr, e
            raise Exception
            return

        udp = ip.child()
        data = udp.get_data_as_string()

        dns1 = ord(data[2])
        dns2 = ord(data[3])

        dnscode = "%x%x" % (dns1, dns2)

        # standard query response, no error
```

```
        if not dnscode == "8180":
            return

        res = tstore.add( hdr, p )

        # if res returns something else than None, a new sample is ready.
        if res:
            eagle.add( tstore.get_data() )
            eagle.preprocess()
            eagle.report()

            eagle.reset()
            tstore.expunge()


    def start(self):
        self.pcap.loop(0, self.packetHandler)

if __name__ == "__main__":
    filename = sys.argv[1]
    p = pcapy.open_offline(filename)

    sample_length = 5
    tstore = TemporalStorage(period=sample_length)
#    tstore = TemporalStorage(period= sample_length , maxsamples=4)
    eagle = iodFinderDST( sample_length )

    # so nasty ;)
    first_ts = None

    try:
        Decoder(p).start()
    except MaxSamplesReached:
        pass
    except KeyboardInterrupt:
        pass

#    import pprint
#    pprint.pprint(eagle.banditstore)

    import os.path
    bname = os.path.basename(filename)
    bname = bname.replace("dnspacketdump-", "")
    bname = bname.replace("-lkarsten.pcap", "")

    print bname,

    # samplelen, falske positives
    print " & %.0f" % sample_length,
    # antall samples foer alarm
    print " & 8 ",
    print " & %.0f" % len(eagle.banditstore),
    print " & X (%s)" % (",".join(eagle.banditstore.keys())),
    print " \\\\ \\hline"


}}
```

Figur B.8: iodFinderDST.py

```
#!/usr/bin/python
```

```
# -*- coding: utf-8 -*-

from iodFinder import iodFinder

class iodFinderDST(iodFinder):
    def __init__(self, slen):
        iodFinder.__init__(self, slen)
        self.persistant_store = [ [], [], [], [], [], [], [], [] ]
        self.tick = 0

        self.banditstore = {}
        self.sample_stop_ts = None

    def analyze(self):
        "Analyze the current sample "
        # def: for aa faa ip-over-dns til aa virke maa man minst
        # faa en dns-forespoersel hvert n-te sekund.
        minbytes = (self.sample_length * 250)
        suspects = []
        # for hver ip observert
        for ip in self.clistore.keys():
            if self.clistore[ip]["queries"] < 10:
                continue
            if self.clistore[ip]["bytes"] < minbytes:
                continue
            if self.clistore[ip]["domain_max_percent"] < 98:
                continue
            suspects.append(ip)
        return set(suspects)

    def timely_analysis(self):
        # shift
        self.tick = self.tick + 1
        for i in range(7, 2):
            self.persistant_store[i] = self.persistant_store[i-1]
        self.persistant_store[1] = self.persistant_store[0]
        self.persistant_store[0] = self.analyze()

        ps = self.persistant_store
        #print "tick %s: %s" % (self.tick, ps)
        bandits = ps[0].intersection(ps[1]).intersection(ps[2])
        bandits = bandits.intersection(ps[3]).intersection(ps[4])
        return bandits

    def report(self, header=True):
#        print bandits
        bandits = self.timely_analysis()

        for ip in bandits:
            for ip in bandits:
                strings = ["ip","domain_max_name"]
                tup = (
                    self.sample_start_ts.strftime("%x %X"),
                    self.clistore[ip]["domain_max_percent"],
                    self.clistore[ip]["bytes"],
                    self.clistore[ip]["domain_max_name"] )

                if not self.banditstore.has_key(ip):
                    self.banditstore[ip] = []
                self.banditstore[ip] += [ tup ]
```

```
}}
```

Figur B.9: realtime-iodFinder.py

```
#!/usr/bin/python
# .: coding: utf-8 :.
#
# apt-get install python-pcapy python-impacket


import os, sys
import pcapy
from impacket.ImpactDecoder import EthDecoder
from impacket.ImpactDecoder import LinuxSLLDecoder, UDPDecoder

from ClientStorage import ClientStorage
from iodClientFinder import iodClientFinderDST, iodClientFinder

class Decoder():
    def __init__(self, pcapObj):
        datalink = pcapObj.datalink()
        if pcapy.DLT_EN10MB == datalink:
            self.decoder = EthDecoder()
        elif pcapy.DLT_LINUX_SLL == datalink:
            self.decoder = LinuxSLLDecoder()
        else:
            raise Exception("Datalink type not supported: " \
                % datalink)

        self.pcap = pcapObj
        self.connections = {}
        self.UDPDecoder = UDPDecoder()

    def start(self):
        self.pcXap.loop(0, self.packetHandler)

def packetHandler(hdr, data):
    global clistore, eagle
    p = EthDecoder()
    p = p.decode(data)

    # hopp over ipv6-pakkene, impacket st0tter det ikke.
    ptype = "%x" % ord(p.get_data_as_string()[0])
    if ptype == "60":
        return

    ip = p.child()
    udp = ip.child()
    data = udp.get_data_as_string()

    dns1 = ord(data[2])
    dns2 = ord(data[3])

    dnscode = "%x%x" % (dns1, dns2)

    # standard query response, no error
    if not dnscode == "8180":
        #print "not a dns response"
        return
```

```
    client = ip.get_ip_dst()
    res = clistore.add( client, hdr, p )

    if res:
        #print "Yielded %s packets for client %s" % (res, client)
        eagle.add( clistore.get_data(client) )
        eagle.preprocess(client)
        eagle.report(client) # , header=False)

        eagle.reset(client)
        clistore.expunge(client)


if __name__ == "__main__":
    config = {
        'domain_max_percent': 98,
        'dev': 'eth1',
        'numpackets': 70,
    }

    config_rates_ssh = {
        # minimum antall bytes per sekund i snitt over samplet.
        'bps_minimum': 560,
        'qps_minimum': 2.27, # empirisk
        'mps_minimum': 140, # empirisk
        'movingaveragedepth': 1,
    }

    ruleset = config_rates_ssh
    for i in ruleset.keys():
        config[i] = ruleset[i]

    # husk "ifconfig eth1 promisc", ellers blir det veldig stille.
    print "opening device %s" % config["dev"]
    p = pcapy.open_live(config["dev"], 1600, 0, 0)
    p.setfilter("udp and port 53")
    p.setnonblock(1)

    clistore = ClientStorage(packets= config["numpackets"])
    eagle = iodClientFinderDST( config )

    try:
        p.loop(-1, packetHandler)
        print "loop returned"
    except KeyboardInterrupt:
        print "Normal exit"
    except Exception, e:
          print e
          import traceback
          traceback.print_exc(5, file=sys.stdout)
    print "dispact returned. Exiting"

}}
```

B.6 Fellesklasser

Figur B.10: TemporalStorage.py

```
#!/usr/bin/python
# .: coding: utf-8 :.

class MaxSamplesReached(Exception):
    pass

class TemporalStorage():
    def __init__(self, period=10, maxsamples=None):
        # period kan vaere None, om man ikke vil at
        # den skal spytte datasett innimellom.
        self.dt = period

        self.last_yield = None

        self.samples_made = 0
        self.samples_max = maxsamples
        # a list of packets.
        self.ds = []
        pass

    def expunge(self):
        #print "expunging %s packets" % len(self.ds)
        self.ds = []

    def get_data(self):
        return self.ds

    def range(self):
        res = (
            self.ds[0][0].getts()[0],
            self.ds[-1][0].getts()[0]
            )
#        print res
        return res

    def add(self, hdr, p):
        self.ds.append( (hdr, p ))
        if not self.last_yield:
            # first packet
            self.last_yield = hdr.getts()[0]

        if self.dt and hdr.getts()[0] > (self.last_yield + self.dt):
            self.samples_made += 1
            self.last_yield = hdr.getts()[0]
            # nasty
            if self.samples_max and self.samples_made >= self.samples_max:
                # control is from Decoder.start(),
                # so this is a bit dirty.
                raise MaxSamplesReached()
            return len(self.ds)
        return None


}}
```

Figur B.11: ClientStorage.py

```
#!/usr/bin/python
# .: coding: utf-8 :.
#
# Ta var paa inntil n pakker fra en klient. Gi beskjed naar taket er naadd.
#

class MaxSamplesReached(Exception):
    pass



class ClientStorage():
    """
    Lagre inntil n pakker for en klient, mens grensesnittet stort sett holdes
    likt TemporalStorage sitt
    """
    def __init__(self, packets=None, bytes=None): #, maxsamples=None):
        # {'clientidentifier': [[list of packets], last_yield], .. }
        self.ds = {}
        pass

    def expunge(self, client):
        if not self.ds[client]:
            raise Exception, 'No data for client seen, unable to expunge'
        self.ds[client][0] = []

    def get_clients(self):
        return self.ds.keys()

    def get_data(self, client):
        return self.ds[client][0]

    def should_yield(self, client):
        raise Exception, "subclassing needed"

    def add(self, client, hdr, p):
        if not self.ds.has_key(client):
            # packetlist, yieldtime
            self.ds[client] = [ [], None]

        self.ds[client][0].append( (hdr, p ) )

        if self.should_yield(client):
            return len(self.ds[client][0])

            #if self.samples_max and self.samples_made >= self.samples_max:
            #    # control is from Decoder.start(), so this is a bit dirty.
            #raise MaxSamplesReached()
        return None

class PacketClientStorage(ClientStorage):
    """
    En ClientStorage som produserer samples hvert n-te pakke
    """
    def __init__(self, packets=50):
        ClientStorage.__init__(self)
        self.maxpackets = packets

    def should_yield(self, client):
        return len(self.ds[client][0]) >= self.maxpackets
```

```
class ByteClientStorage(ClientStorage):
    """
    En ClientStorage som produserer samples etter n byte med pakker
    per klient.
    """
    def __init__(self, bytes):
        ClientStorage.__init__(self)
        self.maxbytes = bytes

    def should_yield(self, client):
        # teller hele pakkestoerrelsen.
        #print len(self.ds[client][0])
        # sannsynligvis urimelig treig, flytt til add-rutina.
        totlen = 0
        for hdr, data in self.ds[client][0]:
            totlen += data.get_size()
            #(hdr, data) = self.ds[client][0][0]
        res = totlen >= self.maxbytes
        return res


}}
```

Figur B.12: iodClientFinder.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

class iodClientFinder():
    """
    Methods to find IP-over-DNS traffic in a
    set of Impacket data packets

    Tar inn samples som bestaar av x pakker. Slik brukt vil ett
    sample kun innholde pakker fra en klient, men samme klienter
    lagres i samme datastruktur.

    Metodene er utvidet til aa akseptere klientidentifikasjon som
    argument.
"""
    import zlib
    from datetime import datetime
    def __init__(self, config):
        #self.reset()
        # iodFinder is used to analyze samples of n seconds worth
        # of packets. Adjust qps/bps values by this to get proper
        # bytes per second.

        self.clistore = {}
        self.config = config
        pass

    def reset(self, client):
        # ts of first data point in sample.
        #raise Exception
        #self.sample_start_ts = None
        if self.clistore.has_key(client):
            del self.clistore[client]
```

```
    def add(self, packetlist):
        if len(packetlist) == 0:
            raise Exception, "tom pakkeliste"

        for hdr, p in packetlist:
            ip = p.child()
            udp = ip.child()
            data = udp.get_data_as_string()
            ipaddr = ip.get_ip_dst()

            #print "adding ipaddr %s" % ipaddr

            labels = self._querylabels(data[12:])
            domainkey = ".".join(labels[-3:])

            tstuple = hdr.getts()
            ts = self.datetime.fromtimestamp(tstuple[0])
            ts = ts.replace(microsecond=tstuple[1])

            if not self.clistore.get(ipaddr):
                self.clistore[ipaddr] = { 'queries': 0,
                            'bytes': 0,
                            'bytes_kolmo': 0,
                            'bytes_kolmo_max': 0,
                            'kolmo_data': [],
                            'mtbq': 0.0,
                            'mtbq_store': [],
                            'domain_store': {},
                            'first_ts': ts,
                            }


            self.clistore[ipaddr]["bytes"] += len(data)
            self.clistore[ipaddr]["queries"] += 1

            # virker unodvendig aa maatte implementere en full
            # dnspakkeleser. sporrefeiltet er ca. ferdig rundt 30
            # byte ut i pakken, og resten er dominert av
            # svardata.
            if len(data) < 35:
                kolmobasis = ''
            else:
                kolmobasis = data[30:]

            self.clistore[ipaddr]["bytes_kolmo_max"] += len( kolmobasis )
            self.clistore[ipaddr]["kolmo_data"] += [ kolmobasis ]

            self.clistore[ipaddr]["mtbq_store"].append( ts )

            try:
                self.clistore[ipaddr]["domain_store"][domainkey] += len(data)
            except KeyError:
                self.clistore[ipaddr]["domain_store"][domainkey] = len(data)

        # sette slutt-tidspunkt for klient i dette samplet.
        self.clistore[ipaddr]["last_ts"] = ts

        # beregne kolmogorov-verdi av konkatineringen av samtlige
        # question-strenger.
        kbasis = "".join(self.clistore[ipaddr]["kolmo_data"])
        kolmobasis_zipped = self.zlib.compress( kbasis )
```

```
        if len(kolmobasis) == 0 or self.clistore[ipaddr]["bytes_kolmo_max"] == 0:
            self.clistore[ipaddr]["kolmoratio"] = 0.00
        else:
            self.clistore[ipaddr]["kolmoratio"] = \
                float( len( kolmobasis_zipped ) ) / \
                self.clistore[ipaddr]["bytes_kolmo_max"]

        del self.clistore[ipaddr]["kolmo_data"]

    def preprocess(self, client):
        # vil bare vaere en, men beholder pga. likhet med
        # iodFinder.
        key = client
        val = self.clistore[client]
        # siste sample kan vaere ganske puny, faa pakker og lite data. ignorer
        # dette i stillhet.
        if val["queries"] < 2:
            return

        for i in range(0, 1):
            val["mtbq_store"].sort()
            val["mtbq_deltas"] = []

            # n^2 is my friend
            for i in range(1, len( val["mtbq_store"])):
                delta = val["mtbq_store"][i] - val["mtbq_store"][i-1]
#                print dir( delta)
                delta_ms = delta.seconds * 1000 + (delta.microseconds / 1000.)
                delta = delta_ms / float(1000)
                #print delta.seconds
                #print delta.microseconds
                val["mtbq_deltas"].append( delta )
#            print "sum: ", sum( val["mtbq_deltas"])
#            print "length: ", len( val["mtbq_deltas"])
            delta_sum = sum( val["mtbq_deltas"] )
            numdeltas = len( val["mtbq_deltas"] )
            if numdeltas == 0:
               val["mtbq"] == None
            else:
                val["mtbq"] = float(delta_sum) / numdeltas

            # mean packet size
            val["mps"] = (float(val["bytes"]) / val["queries"] )

            del val["mtbq_store"]
            del val["mtbq_deltas"]

            #
            val["domain_max_bytes"] = 0
            val["domain_max_name"] = None

            for domain, domainbytes in \
              val["domain_store"].items():
                if domainbytes > val["domain_max_bytes"]:
                    val["domain_max_bytes"] = domainbytes
                    val["domain_max_name"] = domain

            val["domain_max_percent"] = \
                (100 * float(val["domain_max_bytes"]) / val["bytes"])

            sample_length = val["last_ts"] - val["first_ts"]
            #sample_length_ms = sample_length.seconds * 1000 + \
```

```
            #     sample_length.microseconds
            # milli, mikro
            sample_length_ms = (sample_length.seconds * 1000) +
(sample_length.microseconds / float(1000))

            # foreloepig greit aa holde seg paa sekunder, saa
            # faar vi heller leve med avrundingsfeilene.
            val["sample_length"] = float( sample_length_ms ) / 1000.

            val["qps"] = float(val["queries"]) \
                        / val["sample_length"]
            val["bps"] = float(val["bytes"]) \
                        / val["sample_length"]

            del val["domain_max_bytes"]
            del val["domain_store"]

    def report(self, client, header=True):
        keynames = self.clistore.values()[0].keys()
        keynames.pop( keynames.index("last_ts") )
        keynames.pop( keynames.index("first_ts"))

        if header:
            # header
            print "ts\t",
            print "ip\t",
            print "\t".join(keynames)
            #
        # for hver ip observert
        for ip in [ client ]:
            print "%s\t" % \
                self.clistore[ip]["first_ts"].strftime("%X"),

            strings = ["ip",
                "domain_max_name",
                "last_ts","first_ts"]
            print "%s" % ip,
            for key in keynames:
#                print key
                print "\t",
                if key in strings:
                    print self.clistore[ip][key],
                else:
                    print "%.2f" % \
                        float(self.clistore[ip][key]),
            print

    def stats_formatted(self):
        r = self.stats()
        print "%s\t" % self.sample_start_ts,
        print "average\t",
        if False:
            print "%.2f +-%.2f\t" % (r["mtbq"][0], r["mtbq"][1]),
            print "\t\t\t",
            print "%.2f +-%.2f\t" % (r["bytes"][0], r["bytes"][1]),
            print "%.2f +-%.2f\t" % (r["mps"][0], r["mps"][1]),
            print "%.2f +-%.2f\t" % (r["queries"][0], r["queries"][1]),
        else:

            print "%.2f\t" % (r["mtbq"][0]),
            print "\t\t\t",
            print "%.2f\t" % (r["bytes"][0]),
```

```
            print "%.2f\t" % (r["mps"][0]),
            print "%.2f\t" % (r["queries"][0]),

        #print "%.2f (stdev: %.2f)\t" % (r["queries"][0], r["queries"][1]),
        print


    def stats2_header(self):
        r = ["ts"]

        for i in ["mtbq","mps","bps"]:
            for j in ["iodine","mean", "stdev"]:
                r.append( "%s_%s" % (i, j))

        r += ["numclients"]
        r += ["qps_per_cli"]
        return r
        #print "\t".join(r)

    def stats2(self):
        "tid unike_klienter iodine_bytes iodine_x i_y, avg_foo avg_bar, avg_qux"

        r = self.stats()
        iod = self.clistore.get("129.241.208.220") or {}

        res = [ "%s" % self.sample_start_ts ]

        for i in ["mtbq","mps","bps"]:
            iodine_value = iod.get(i)
            if not iodine_value:
                iodine_value = "NA"
            else:
                iodine_value = "%.2f" % float(iodine_value)
            res.append( "%s" % iodine_value)
            res.append( "%.2f" % r.get(i)[0] )
            res.append( "%.2f" % r.get(i)[1] )

        # numclients og qps_per_cli
        res += [ str(len(self.clistore)) ]
        qps_per_cli = r.get("qps")[0] / float(len(self.clistore))
        res += [ "%.6f" % (qps_per_cli) ]

        return res
        #print "\t".join(res)

    def stats(self):
        """Find sample statistics

        mean
        stdev

        of:
            mean packet size.
            mean time between queries.
            numqueries
            numbytes

        """
        from statlib import stats
        keys = ["mps","mtbq","qps","bps"]
        res = {}
        for name in keys:
```

```
            # dra ut data. litt hummer&kanari om det er string
            # eller ei, saa cast alt til float.
            vector = [ float(i[name]) for i in self.clistore.values() ]

            if len(vector) == 0:
                raise Exception, "No values found for key %s" % name

            elif len(vector) == 1:
                res[name] = ( vector[0], 0.0 )
            else:
                # median? quantiles?
                res[name] = ( stats.mean( vector ), stats.stdev( vector) )
        return res


    def _querylabels(self, data):
        """Return a list of the labels in the query portion of a
        DNS packet"""
        i = 0
        labels = []
        while True:
            # print querydata[i]
            llen = int("%02x" % ord(data[i]), 16)
            #print type(llen), llen

            if llen == 0:
                break

            s1 = i+1
            s2 = s1 + llen
            label = data[s1:s2]

            labels.append(label)
            i = i + llen +1

        return labels


class iodClientFinderDST(iodClientFinder):
    def __init__(self, config):
        iodClientFinder.__init__(self, config)
        self.persistant_store = {}
        self.tick = {}

        self.banditstore = {}
        self.sample_stop_ts = None

    def analyze(self, client):
        """
        Gir denne klienten noen advarsel/alarm i dette samplet?
        returnerer True, false.

        Analyze the current sample """
        verbose = True
        veryverbose = True

        data = self.clistore[client]

        # tester som helt klart sier at det IKKE skal gis alarm,
        # men som ikke sier noe om det skal gis alarm.
        if data["bps"] < self.config["bps_minimum"]: #minbytes:
            if veryverbose:
```

```
            #self.pprint(data)
            print "non-iod, low data: %s below configured %s" % \
                (data["bps"], self.config["bps_minimum"])
        return False

    if data["qps"] < self.config["qps_minimum"]:
        if veryverbose:
            print "non-iod, qps of %s below required/configured %s" % \
                (data["qps"], self.config["qps_minimum"])
        return False

    if data["mps"] < self.config["mps_minimum"]:
        if veryverbose:
            print "non-iod, mean packet size of " + \
               "%s below required/configured %s" % \
               (data["mps"], self.config["mps_minimum"])
        return False


    final_result = False
    # tester som sier om det kan vaere iod.
    if data["domain_max_percent"] > self.config["domain_max_percent"]:
        if verbose:
            print "domain_max_percent alarm: " + \
                "%s more than configured %s" % \
                (data["domain_max_percent"], self.config["domain_max_percent"]
        final_result = final_result or True

    if final_result is True:
        print self.pprint(data)
        return final_result
    return final_result

def timely_analysis(self, client, depth, verbose=False):
    """
    Analyse per klient, over tid. Om en klient sitt datasett
    gir alarm, vil det lagret her.  Maalet er aa se etter
    etterfoelgende alarmer, saa det vil fungere aa gruppere per
    bruker
    """
    # shift
    if not self.tick.has_key(client):
        self.tick[client] = 0
    self.tick[client] = self.tick[client] + 1
    #for i in range(7, 2):
    #if depth > 8:
    #    raise Exception, "not supported"

    if not self.persistant_store.has_key(client):
        # n last lists of suspects.
        self.persistant_store[client] = \
            [None]* depth

    ps = self.persistant_store
    c = client

    if depth == 1:
        return self.analyze(c)

    if verbose:
        print "pretick %-20s %s: %s" % \
           (client, self.tick[client], ps[client])
```

```
        # shift right
        # hva om den er to?
        if depth >= 2:
            for i in range(depth-1, 0, -1):
                #print "shift %s" % i
                self.persistant_store[c][i] = self.persistant_store[c][i-1]

        self.persistant_store[c][0] = self.analyze(c)

        if verbose:
            print "   tick %-20s %s: %s" % \
               (client, self.tick[client], ps[client])

        # AND sammen listen over true/false paa de siste n
        # samples.
        warnings = ps[c][0]
        for i in range(0, depth):
            warnings = warnings and ps[c][i]
        #print "final: ", warnings
        return warnings # alarm

    def pprint(self, d):
        import pprint
        pprint.pprint(d)

    def report(self, client, header=True):
        raise Exception, "Skal subklasses"
        alarm = self.timely_analysis(client, \
            self.config["movingaveragedepth"], verbose=True)
        if not alarm:
            return None

        print "alarm: %s " % (client)

        strings = ["ip","domain_max_name"]
        if not self.banditstore.has_key(client):
            self.banditstore[client] = []
        self.banditstore[client] += [ self.clistore[client] ]

        r = "Probable IP-over-DNS client:" + \
            "%s (%.0f%% of %.0f bytes from domain %s)" % \
                (
                client,
                self.clistore[client]["domain_max_percent"],
                self.clistore[client]["bytes"],
                self.clistore[client]["domain_max_name"],
                ),
        print " ".join(r)


}}
```

Figur B.13: EagleReporter.py

```
#!/usr/bin/python
# .- coding: utf-8 -.
#
# Rapporteringsrutine for dict(iodClientFinder.bandits)

import os, sys
```

```
from ClientStorage import ClientStorage, PacketClientStorage
from ClientStorage import ByteClientStorage
from iodClientFinder import iodClientFinder
from iodClientFinderKolmo import iodClientFinderKolmo

class EagleReporter:
    def __init__(self, config, filename, bandits, veryfirstts):
        self.config = config
        self.filename = filename
        self.bandits = bandits
        self.veryfirstts = veryfirstts

    def hent_positiver(self, fname=None):
        """
        Hardkodet liste over kjente positiver i datafilene,
        slik at rapporteringen kan gjoeres automatisk.
        """
        if not fname:
            fname = self.filename
        positiver = {"dnspacketdump-2010-04-27-lkarsten.pcap": set(["129.241.208.220"]) }

        spesielle = ["iodine-login.pcap", "iodine-poll.pcap",
        "iodine-mp3stream.pcap", "iodine-ssh.pcap",
        "iodine-web.pcap", "nstx-login.pcap", "nstx-poll.pcap",
        "nstx-ssh-mtu500.pcap", "nstx-ssh-mtu1500.pcap",
        "nstx-web.pcap", "tuns-login.pcap", "tuns-poll.pcap",
        "tuns-ssh.pcap", "tuns-web.pcap"]

        for i in spesielle:
            positiver[i] = set(["10.51.1.140"])

        res = positiver.get(fname )
        if not res:
            res = set([])
        #print "Filen %s har %s kjente positiver: %s" % \
        # (fname, len(res), ",".join(res))
        return res

    def report(self):
        raise Exception, "subclass"


class EagleLatexReporter(EagleReporter):
    def report(self):
        bname = self.filename
        fname = os.path.basename(self.filename)
        kjente_pos = self.hent_positiver(fname=fname)

        import pprint
        #pprint.pprint(eagle.banditstore)

        # friske opp utskriften litt
        bname = bname.replace("dnspacketdump-", "")
        bname = bname.replace("-lkarsten", "")
        bname = bname.replace("../datafiles/", "")
        bname = bname.replace(".pcap", "")
        print bname,

        print "& %.0f" % self.config["numpackets"],
        print "& %.2f" % self.config["kolmo_minimum"],
```

```
        # positiver
        pos = set(self.bandits.keys())
        # et sett med ipadressene til falske positiver
        falskpos = pos - kjente_pos
        #print " pos:  %.0f & " % len(pos),
        # manglende kjente positiver. (falske negativer)
        fneg = kjente_pos - pos
        print "& %.0f av %i kjente (%i falske, %i mangler)" % \
                (len(pos), len(kjente_pos), len(falskpos), len(fneg)),

        print "& ",
        for ip in kjente_pos - fneg:
            m = self.bandits[ip][0]
            # alarmen gaar naar samplet er slutt, og blitt regnet paa.
            delta_t = m.get("last_ts") - self.veryfirstts
            delta_t_float = delta_t.seconds + (delta_t.microseconds / 1000000.)
            print "%0.2fs " % (delta_t_float),
        print " \\\\ \\hline"


}}
```