Selvorganiserende læring av trafikkategorier i Bayesiansk pakkebasert IDS

av

Tor Oskar Wilhelmsen

Masteroppgave i
informasjons- og kommunikasjonsteknologi

Høgskolen i Agder
Fakultet for teknologi

Grimstad
mai 2005

1 Sammendrag

Detektering av uønsket trafikk i et datanettverk utføres i dag av systemer med mer eller mindre grad av manuelt vedlikehold. Slikt vedlikehold består typisk av nedlasting av oppdateringer som bidrar til deteksjon av de nyeste typene angrep. For å unngå dette manuelle vedlikeholdet forskes det rundt systemer som skal være i stand til å lære seg hva som er normal og hva som er unormal nettverkstrafikk i et datanettverk. Slike systemer er gjerne basert på en læringsalgoritme, som basert på et sett med treningsdata, kan ta beslutninger om hva som er normalt og ikke.

Denne masteroppgaven ser på hvordan en enkel læringsalgoritme, den Naive Bayesianske klassifikatoren, greier oppgaven med å klassifisere normal og unormal nettverkstrafikk. For å oppnå best mulig resultater fokuserer vi mye på hvordan vi skal velge attributtene til den Naive Bayesianske klassifikatoren. Vi foreslår fire forskjellige måter for attributtutvelgelse. En av metode baserer seg på å bruke feltene i IP og TCP/UDP protokollene som attributter, en metode benytter bitene i pakken og grupperer dem sekvensielt i 8 og 8 bits mens de 2 siste ser på relasjonene mellom bits i pakken for å gruppere attributtene. For å beregne relasjonene mellom bits baserer vi oss på en generell algoritme for å beregne avhengigheten mellom variabler i en datamengde.

De ulike metodene blir evaluert ved å bruke Darpa datasettet fra MIT, og resultatene viser at vi er i stand til å detektere 99,89 % av den normale nettverkstrafikken ved å bruke attributter med variabel lengde som er satt sammen av bits som er i relasjon til hverandre. Falske alarmer vil da utgjøre 0,11 % av antall klassifiserte pakker noe som betyr at antallet falske alarmer pr dag med klassifisert trafikk er overkommelig.

Av angrep greier vi å detektere 13 av 16 angrepsinstanser av typen DOS, U2R, R2L og Probe, men dette fører med seg at det blir en økning i antall falske alarmer. For å senke antall falske alarmer samtidig som vi opprettholder evnen til å detektere angrep ser vi på hvordan forholdet mellom falske alarmer og feilklassifiseringer kan optimaliseres.

2 Forord

Rapporten er skrevet som en del av Masterutdanningen i informasjons og kommunikasjonsteknologi(IKT) ved Høgskolen i Agder, avdeling for teknologi i Grimstad. Arbeidet tilsvarer 30 studiepoeng og har pågått fra januar til juni 2005.

Jeg vil få takke min veileder ved Høgskolen i Agder, Ole-Christoffer Granmo, for mange gode innspill og god faglig støtte gjennom hele oppgaven.

Tor Oskar Wilhelmsen

3 Innholdsfortegnelse

3.1 Figurliste

3.2 Tabelliste

4 Innledning

4.1 Bakgrunn for oppgaven

Utviklingen i de senere år har ført til at stadig flere datamaskiner er koblet sammen i nettverk, som for eksempel Internett. Internett er en samling av flere nettverk som er koblet sammen ved hjelp av kraftige backbone nettverk som sørger for høyhastighetskommunikasjon mellom de sammenkoblede nettverkene. Stadig større nettverk, med økende antall maskiner er en utfordring når det gjelder å beskytte nettverk og maskiner mot innbrudd og destruktive angrep. En vanlig måte å beskytte nettverk og enkeltmaskiner mot angrep på er å benytte en brannmur. En brannmur sperrer for uønsket trafikk ved å sperre porter og adresser for innkommende eller utgående trafikk. De fleste brannmurer er derimot ikke utviklet for å kjenne igjen uønsket trafikk til de åpne portene. Eksempel på slik uønsket trafikk er DOS (Denial of service) angrep. I et DOS angrep sendes det store mengder pakker, eksempelvis mot en webserver, som enkeltvis er helt ufarlige. Pakkene vil slippe igjennom de åpne portene i brannmuren og vil sørge for at webserveren i ytterste konsekvens vil slutte å respondere. Dette kan være uheldig i tilfellene der webserveren er forretningskritisk. Andre eksempler på uønsket trafikk kan være datapakker som sørger for å utnytte svakheter i PC-ens programvare til å overta kontrollen over PC-en med den hensikt å stjele informasjon eller utføre videre innbrudd i andre systemer.

Skal slike angrep oppdages er det nødvendig å analysere den innkommende pakkestrømmen for å se om den innholder uønsket nettverkstrafikk av noe slag. Et program for å analysere og detektere uønsket datatrafikk betegnes ofte som et innbruddsdeteksjonssystem, IDS (Eng: Intrusion Detection System). Slike systemer finnes på markedet i dag, men felles for mange av dem er at de baserer seg på å gjenkjenne angrep ved hjelp av en signatur for angrepet. Signaturen er en beskrivelse av et mønster som er typisk for et spesielt angrep. En slik signatur må produseres for hvert angrep som skal kunne detekteres. Dette betyr at IDS'en må oppdateres manuelt med den nye signaturen, noe som betyr at IDS'en er sårbar for angrep i tidsrommet fra et angrep dukker opp til IDS'en er oppdatert.

Ved å se problemet fra et litt annet synspunkt kan vi la IDS'en prøve å finne ut hva som er normaltrafikk rapportere om alt som ikke er normalt. Skal en slik fremgangsmåte være mulig må denne metoden ta i bruk en eller annen form for maskinlæring slik at vi kan greie å lære maskinen til å ta "sine egne valg". Det har vært gjort ulike forsøk, tildels med gode resultater, der ulike læringsteknikker har blitt brukt. Det er imidlertid fortsatt stort potensial for forbedringer både med tanke på bedre deteksjon og med tanke på å senke antall falske alarmer.

I denne oppgaven bruker vi noen teknikker fra liknende arbeid sammen med noen nye ideer om hvordan attributter skal velges for å se om vi kan bruke en Naiv Bayesiansk klassifikator til å klassifisere nettverkstrafikk på en tilfredsstillende måte. En av de store fordelene med

den Naive Bayesianske klassifikatoren er at trening og klassifikasjon skjer ved en enkelt iterasjon gjennom trenings / testdata noe som gjør den effektiv med hensyn til prosesseringstid i forhold til en del andre algoritmer som trenger flere iterasjoner gjennom trenings og testdata.

4.2 Beslektet arbeid

Det er utført en del arbeid som omhandler innbruddsdeteksjonssystemer, som alle har til hensikt å utvikle metoder for deteksjon av unormal nettverkstrafikk. Noe av dette arbeidet baserer seg på ”deteksjonsmotorer” som bruker manuelt genererte signaturer for så å søke etter disse i pakkehodet og pakkeinnholdet. En slik signatur er typisk en beskrivelse av hvilke pakker et angrep inneholder samt rekkefølgen på pakkene. Ved match på den genererte signaturen vil vi da få detektert unormal trafikk. Systemer som bruker denne metoden er Bro[5] og SNORT[6]. Dette er eksempler på systemer som trenger manuell oppdatering, og som derfor vil være sårbare i tiden fra et angrep er oppdaget til det er utviklet en signatur og systemet er oppdatert.
I motsetning til disse har vi systemer som SPADE[7], ADAM[8] og NIDES[9] som bygger opp en statistisk modell fra angrepsfri nettverkstrafikk. For å bygge denne statistiske modellen benyttes kilde og destinasjonsadresser samt porter som brukes i en sesjon. Trafikk som avviker fra den bygde modellen vil bli merket som unormal. Hvis vi tar for oss SPADE ser vi at den beregner følgende:

- P(destinasjonsadresse, destinasjonsport)
- P(kildeadresse, destinasjonsadresse, destinasjonsport)
- P(kildeadresse, kildeport, destinasjonsadresse, destinasjonsport)
- Beregning av Bayesiansk nett av de foregående

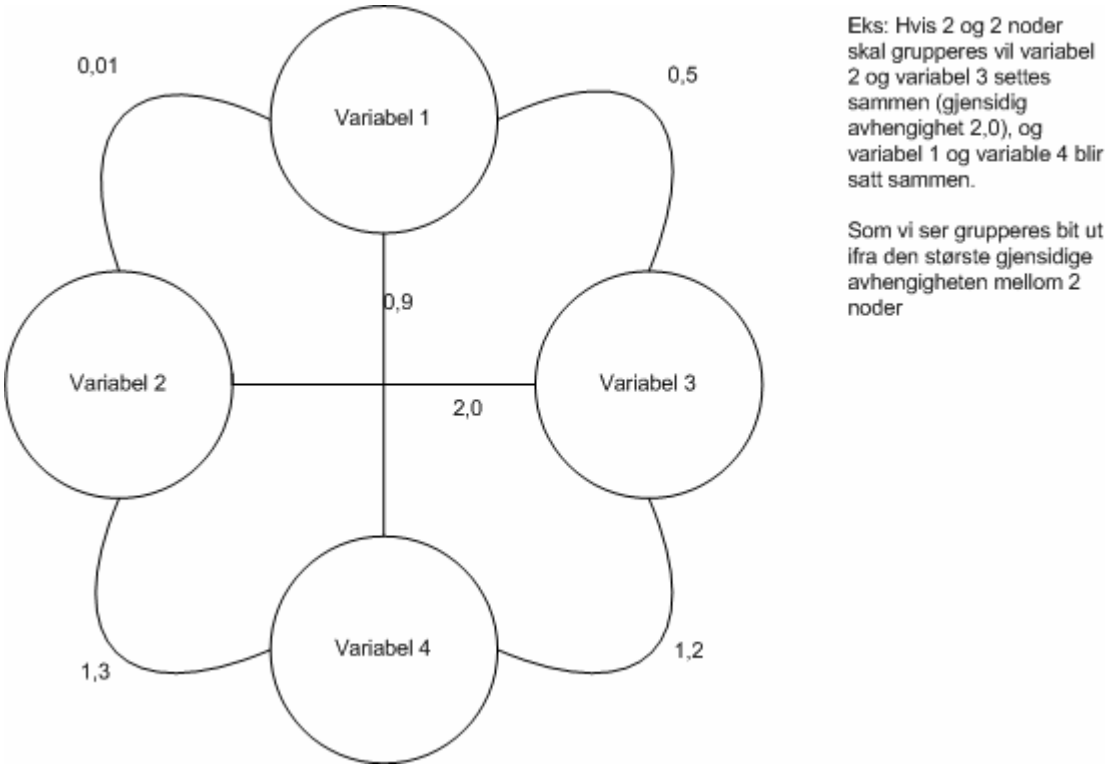
Der vi med P mener sannsynligheten.
Felles for SPADE, ADAM og NIDES er at de baserer seg på en frekvensmodell som bygges opp ved å se på frekvensen av uregelmessigheter i treningsdataene. Sannsynligheten for at en uregelmessighet oppstår vil da bli beregnet ut fra gjennomsnittelig frekvens av uregelmessigheten i treningsdataene.
Andre systemer som også bruker statistikk og sannsynlighetsregning, men baserer en sannsynlighetsverdi med hensyn på hvor lang tid det har gått siden sist uregelmessighet, er PHAD[10], ALAD[12], LERAD[11] og NETAD[1]. Hvert attributt tilegnes et sett av lovlige verdier, slik at en ny og ukjent verdi for et attributt fører til en alarm. Hovedforskjellen på metodene som baserer seg på en tidsmodell, er attributtene som brukes. PHAD baserer seg på protokollfeltene i pakkehodet (34 attributter i alt), ALAD bygger sin modell på en innkommende TCP-connection, og bruker feltene kildeadresse, destinasjonsadresse, kilde og destinasjonsporter, tcp-flag og listen av kommandoer selve pakkedataene. LERAD bruker den samme formen for modell som ALAD, men i tillegg samples treningsdata for å lage regler. Et eksempel på en slik regel er hvis det i treningsdataene kun er 2 http tilknytninger til en maskin sier LERAD at det bare er lovlig med http trafikk til denne maskinen.

I NETAD modellen innføres det en flertrinnsprosess for å klassifisere trafikken. Det første trinnet er et filter som tar bort trafikk som ikke vil ha særlig innvirkning på deteksjon av uregelmessigheter. Denne trafikken er:

- Ikke IP trafikk (ARP osv).
- All utgående trafikk.
- All TCP trafikk som starter med en SYN-ACK pakke, noe som indikerer at pakken er initiert lokalt. Normalt initieres angrep utenfra mot en local server.
- UDP trafikk til porter >1023.
- TCP trafikk som starter etter at de første 100 bytes er overført (iflg. sekvensnummeret).
- Pakker til en adresse/port/protokoll kombinasjon der det har kommet mer enn 16 pakker i løpet av 60 sekunder.
- Pakkestørrelsen trunkeres til 250 bytes, selv om NETAD bruker kun 48 bytes

Det neste trinnet i NETAD består av en klassifikator. Klassifikatoren bruker 48 attributter for klassifisering. Hver av de 48 første bytene (40 byte i pakkehode + 8 byte av pakkeinnhold) er et eget attributt som kan ha opp til 256 forskjellige verdier. NETAD bruker i første omgang klassifikatoren til skille på klassene angrep eller normaltrafikk. Modellen bruker en ”Scoring function” for å måle ytelsen ved forskjellige terskler av falske alarmer. Scoring funksjonen er gitt som $\Sigma\, tn_a(1 - r/256) / r + t_i / (f_i + r / 256)$. Den første delen, $tn_a(1 - r/256) / r$, tar hensyn til hvor lang tid det er siden sist attributtet forekom. Den siste delen, $t_i / (f_i + r / 256)$, tar hensyn til frekvensen av attributtverdien i treningsdataene.

Når vi har store datamengder som skal klassifiseres og trenes, vil det alltid være noe informasjon i denne datamengden som er mer avhengig av hverandre enn andre. I ”An Accelerated Chow and Liu Algorithm”[4] viser Marina Meila en generell metode for å gruppere informasjon som er gjensidig avhengig. Denne algoritmen baserer seg på en algoritme utviklet av Chow og Liu i 1968 (CL algoritmen). Den akselererte CL algoritmen en raskere versjon av CL algoritmen, slik at den effektivt kan brukes til å beregne avhengigheter i en datamengde.
Det grunnleggende prinsippet i algoritmen er å beregne i hvilken grad 2 attributter i datamengden er avhengig av hverandre. Denne informasjonen brukes til å lage et tenkt tre, der hver variabel i datamengden er en node, mens avhengigheten mellom nodene beskrives som en kant(egde) mellom nodene.



Figur 1. Eksempel på graf med avhengigheter mellom variabler.

Denne informasjonen kan igjen brukes til å trekke ut viktig informasjon fra datamengden. I store datamengder vil det være en fordel å kunne skille ut og ignorere informasjon som ikke har noe betydning for videre behandling av data.

4.3 Hypoteser

Oppgaven har flere elementer og de hypotesene vi ønsket å undersøke var:

1. *Den Naive Bayesianske klassifikatoren kan ved riktig valg av attributter brukes som læringsalgoritme i et innbruddsdeteksjonssystem.*
2. *Det vil være stor forskjell på ytelsen til den Naive Bayesianske klassifikatoren avhengig av hvordan vi grupperer attributtene*

4.4 Rapportens bidrag

I denne oppgaven ser vi spesielt på hvordan vi kan velge ut attributter for bruk i en Naiv Bayesiansk klassifikator, og hvordan ytelsen varierer med hva slags attributter som velges. Måtene å velge attributter på spenner fra manuell utvelgelse av attributtene til beregning av optimale attributter ved å se på hvilke bits i en pakke som er mest relatert til hverandre for dermed å gruppere disse i samme attributt. Disse ulike måtene å velge attributter på blir målt ved hjelp av tester på Darpa datasettet fra MIT.

4.5 Rapportens oppbygging

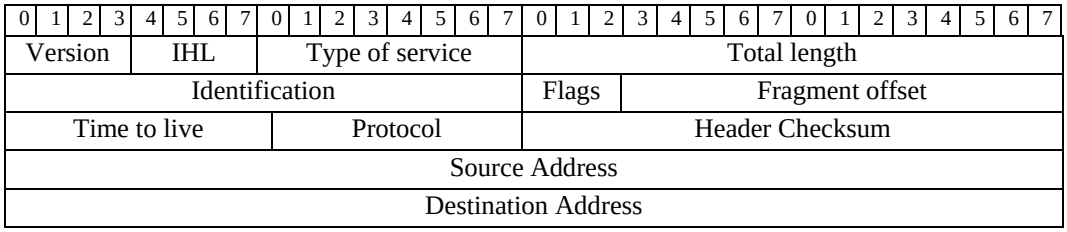
Rapporten vil videre i kapittel 5 ta for seg en introduksjon til IP nettverkstrafikk, Darpa datasettet og naiv Bayesiansk klassifikator. Dette kapittelet innholder grunnleggene informasjon for videre forståelse av metoder og resultater som presenteres. Emne IP dekker også en kort introduksjon til transportlagsprotokollen TCP og UDP samt en introduksjon til støtteprotokollen ICMP.
I kapittel 6 gir vi en gjennomgang av de fem ulike måtene vi har valgt attributter til klassifikatoren på. Kapittel 7 gir oss resultatene for de ulike måtene vi har valgt attributter på, mens vi i kapittel 8 drøfter disse resultatene.
Kapittel 9 inneholder oppgavens konklusjon mens litteraturliste og vedlegg følger i kapittel 10 og 11.

5 Introduksjon til nettverkstrafikk, Darpa datasett og naiv Bayesiansk klassifikator

I dette kapittelet gis det korte introduksjoner til emner som har betydning for videre forståelse av rapporten. Vi tar opp emner som IP, TCP, UDP og ICMP for å gi en grunnleggende forståelse av protokollens headere for bedre å forstå utvelgelse av attributter. Videre blir Darpa datasettet beskrevet slik at det skal skapes en forståelse av hva slags data resultatene blir generert på grunnlag av. Til slutt introduserer vi den Naive Bayesianske klassifikatoren og gir et eksempel på hvordan denne kan brukes. Emnene behandles ikke i sin helhet, men de felter av emnene som oppgavene bygger på blir nevnt.

5.1 IP

I et datanettverk forgår kommunikasjonen mellom de ulike nettverksenhetene (som PC, ruter osv) ved hjelp av pakkesvitsjet kommunikasjon. Et pakkesvitsjet nettverk er et såkalt forbindelsesløst nettverk som betyr at det vanligvis ikke settes opp faste forbindelser mellom sender og mottaker før kommunikasjonen begynner. I stedet sendes det pakker, som ved hjelp



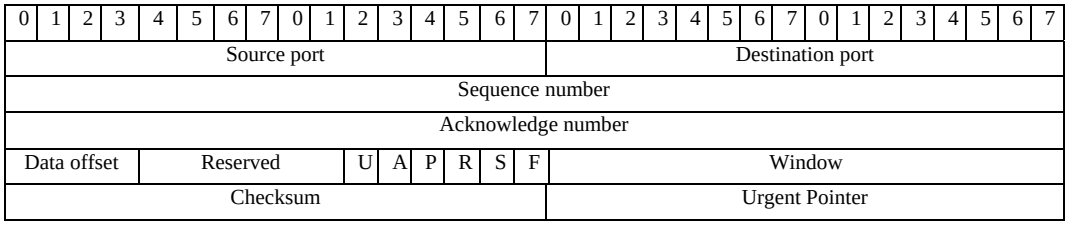
Figur 2. Feltene i IP protokollens pakkehode

av en adresse som settes på pakken, dirigeres frem til mottaker. For at en pakke skal dirigeres til riktig destinasjon må den være på et forhåndsdefinert format. Dette formatet er definert i en såkalt protokoll. Per i dag er IP (Internet Protocol) den mest brukte protokollen for transport av pakker i et datanettverk. IP er en standardisert protokoll som beskriver hvilket format en pakke må ha for å kunne sendes gjennom nettverket. IP sier at hver pakke skal ha en header, som er metadata om pakken. Hver pakke kan bestå av opp til 1500 bytes (inkludert header) med nyttedata. Hvis datamengden som skal sendes over nettverket er større enn 1500 bytes blir pakken delt opp, og de ulike fragmentene nummereres og sendes i hver sin pakke slik at mottakeren kan sette dem sammen i den rekkefølgen de opprinnelig hadde. Figur 2 skisserer formatet til IP headeren. Alle feltene i IP headeren har en mening, men de som har mest betydning for at pakken skal komme frem til riktig mottaker er kildeadresse og destinasjonsadresse. Disse feltene inneholder IP-adressen, og et eksempel på en IP adresse er 192.168.0.1. Disse adressene kan igjen deles inn i nettverksadresse og maskinadresse, og da vil ofte adressen skrives 192.168.0.1/24 der vi ser adressen og tallet bak / sier hvor mange bits av adressen som er nettverksadressen. De resterende bitene brukes som maskinadresse. Denne

inndelingen, som betegnes subnetting, brukes for å effektivt kunne utnytte adressene som eksisterer i dagens versjon av IP(versjon 4). IP versjon 6, som med tiden antageligvis vil bli innført, bruker 128 bits i IP-adressen, og det vil dermed ikke lenger være behov for subnetting på grunn av at det vil være nok adresser til at alle enheter som kobles til et datanettverk vil kunne få sin egen unike IP adresse.
IP er en transportprotokoll for andre protokoller. Det finnes også andre slike transportprotokoller, som ATM, men IP er den mest brukte. Alle andre kjente høyere lags protokoller kan bli transportert over IP.

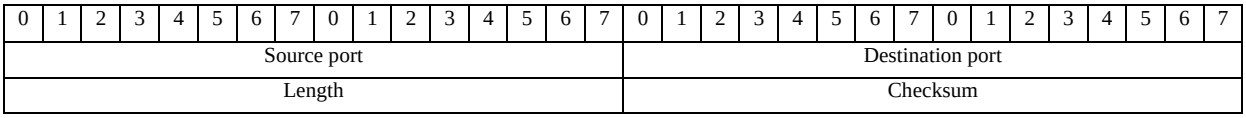
5.2 TCP og UDP

IP er en transportprotokoll som ikke blir brukt frittstående, men som transportør av andre protokoller. De mest aktuelle er TCP (Transmission Control Protocol) og UDP (User Datagram Protocol). Dette er protokoller som sørger for at applikasjoner kan ”snakke sammen”. Mens IP sørger for å transportere pakken til riktig adresse, vil TCP og UDP sørge for å bringe pakken til riktig port. En port kan vi se på som et aksesspunkt mot en applikasjon som kjører på en datamaskin, og en applikasjon kan gjerne benytte flere porter. Det betyr at en PC kan kommunisere med flere andre PC-er på en adresse, men med 1 port åpen for hver forbindelse.



Figur 3. Feltene i TCP pakkens hode

Den største funksjonelle forskjellen på TCP og UDP er at TCP tilbyr pålitelig kommunikasjon, mens UDP sender pakker uten garanti for at mottakeren får pakkene.
En TCP forbindelse blir satt opp ved en såkalt 3-way-handshake. Dette innebærer at en maskin initierer en forbindelse. En pakke som initierer en forbindelse har satt SYN flagget (merket som S i pakkehodet i Figur 3). Maskinen som det ønskes kontakt med svarer med en SYN-ACK pakke (SYN og ACK flagg er satt), og den initierende maskinen svarer så med en ACK pakke. Hvis denne prosessen fullføres vil det være satt opp en forbindelse som sikrer pålitelig overføring av data. I TCP-headeren har vi i tillegg til kilde port og avsendeport en del felt som skal sikre denne pålitelige overføringen. Dette gjøres ved at hver pakke som sendes får et sekvensnummer, der mottakeren svarer med et ”ack”-nummer på at pakken er mottatt. Dette sørger for at pakker som ikke er meldt mottatt, blir sendt om igjen inntil pakken er bekreftet mottatt. Headeren inneholder også mekanismer for å regulere mengden av pakker som sendes i nettet slik at vi unngår metning i nettverket og tap av pakker, men dette kommer jeg ikke videre inn på her, da dette er utenfor oppgavens omfang.



Figur 4. Feltene i UDP pakkehodet

UDP er en mye enklere protokoll som gir større hastighet og mindre overhead på bekostning av et potensielt tap av pakker. Pakketap kan forekomme hvis det for eksempel sendes mange flere pakker enn ruterne i nettet greier å håndtere. UDP merker kun en pakke med portnummeret pakken skal til og portnummeret pakken kommer fra før den transporteres av IP protokollen. UDP har ingen pålitelighetskontroll eller metningskontroll slik som TCP.

5.3 ICMP

I tillegg til TCP og UDP protokollene som er aktuelle i denne rapporten, har vi ICMP. ICMP er en støtteprotokoll for IP protokollen. Denne kan brukes til å sende meldinger som er av betydning, f. eks for rutingen av en datapakke. ICMP brukes ikke for transport av nyttedata slik som TCP og UDP.

5.4 Darpa datasett

For å kunne sammenlikne ytelse og resultater fra ulike IDS'er er det nødvendig med et felles grunnlag for trening og testing av de utviklede IDS'ene. Dette behovet resulterte i et DARPA datasett [3], som var grunnlag for en "Darpa Offline Intrusion Detection Evaluation"[2] der ulike IDS'er ble evaluert opp mot hverandre. I denne forbindelse har Lincoln Laboratory ved Massachusetts Institue of Technology (MIT) utviklet et datasett som skal være en referanse for å måle ytelsen på et IDS. Datasettet består av treningsdata og testdata. Treningsdataene er nettverkstrafikk som inneholder markerte angrepssesjoner. På denne måten kan angrepsdata skilles ut og brukes i treningssammenheng. Testdata er bygd opp på samme måte, men det var først i ettertid at det ble gitt ut en oversikt over angrepene i testdataene.
Det ble gitt ut et slikt datasett i 1998 og ytterligere et i 1999. I datasettet fra 1999 er det vanskeligere å skille ut angrepsdata eksplisitt, noe som har ført til at denne oppgaven bygger på datasettet fra 1998.
Darpa datasettet deler opp angrepstrafikken i klasser, avhengig av hva slags type angrep det er. Disse klassene er:

- DOS: Denial of Service, som er angreps som går ut på å nekte en maskin (server) og utføre sine oppgaver. Disse angrepene utføres ofte ved å sende så mange forespørsler til en server at serveren ikke greier å håndtere disse forespørslene og bryter sammen.
- U2R: User to Root, som er at angrep som har til hensikt å overta en maskin ved å få tilgang til root kontoen på maskinen. Dette gjøres ved å utnytte svakheter i operativsystemet, slik at inntrengeren får full kontroll fra maskinen.

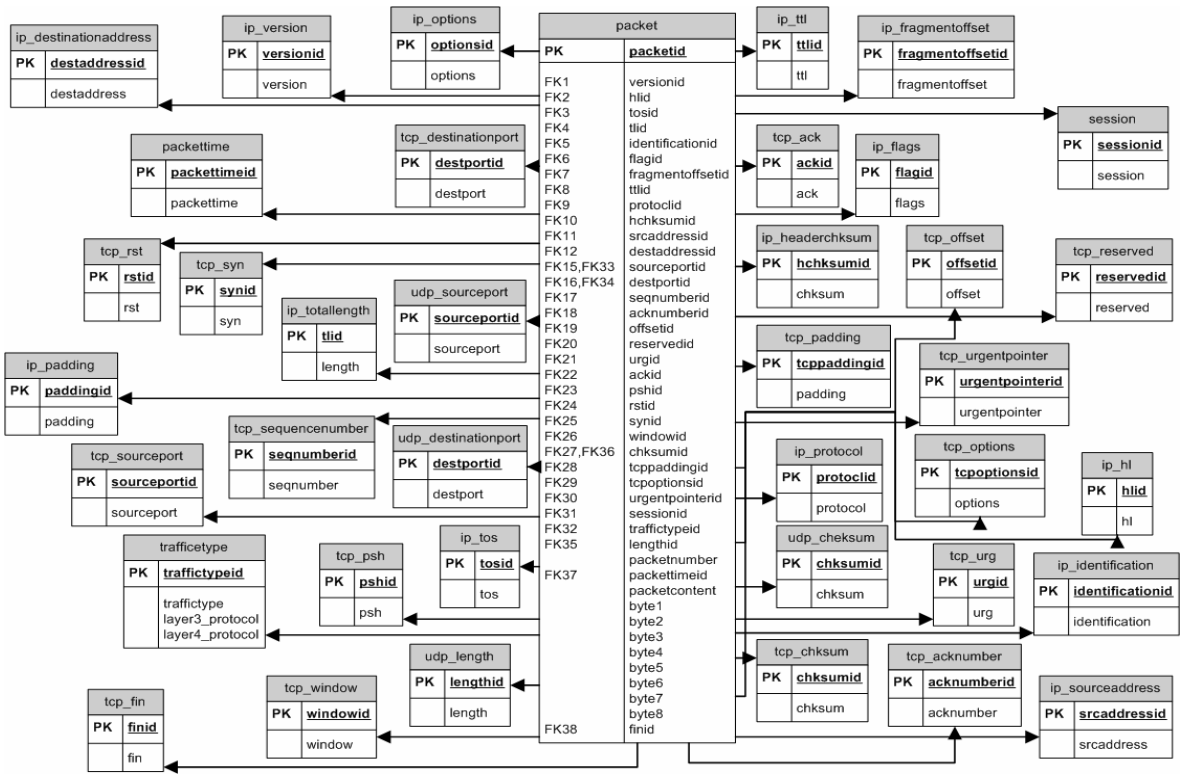
- PROBE: Probe angrep prøver å kartlegge hvilke tjenester og åpne porter som er åpne på en maskin, typisk ”port scan”. Et slikt angrep forekommer ofte i forkant av andre angrep, og er ikke farlige i seg selv.
- R2L: Remote to local angrep prøver å utnytte svakheter i nettverksprogrammer til for eksempel å laste inn ”spyware” og lignende programsnutter på en maskin.
- DATA: Prøver å utføre ulovlige operasjoner i følge systemets sikkerhetspolitikk. Denne angrepsklassen eksisterer ikke 1998-datasettet.

Disse angrepsklassene står sentralt i min oppgave, da jeg i tillegg til å se på hvordan alle angrep skal detekteres, prøver å plassere angrepene i den klassen de høre hjemme.

De forskjellige klassene inneholder igjen flere ulike angrep. Det er flere ulike måter å gjennomføre et angrep, hvor det eneste de har til felles er at de ønsker å være mest mulig likt normal angrepsfri trafikk. Angrepene varierer i omfang og størrelse. Mens de minste instansene av angrep er 1 pakke, består noen angrep av opptil flere hundretusen pakker. Det blir også brukt ulike protokoller for angrep innen en trafikkklasse. Et eksempel er en type DOS angrep hvor en rekke TCP pakker som sendes mot en server eller det kan være en serie ICMP pakker som serveren må behandle. Felles for dem begge er at serveren blir overarbeidet på grunn av for stor pågang av innkommende pakker.

Treningsdataene som brukes videre i oppgaven er de tre første ukene med data fra Darpa datasettet. Dette er tcpdump-filer som emulerer virkelig nettverkstrafikk. I disse filene vil normal trafikk og angrepstrafikk komme om hverandre, noe som ikke er hensiktsmessig for å trene opp ulike trafikklasser. Det mest hensiktsmessige når det kommer til opptrening er å kunne trene hver trafikklasse for seg selv. For å kunne gjøre dette er det opprettet en database som lagrer alle treningsdata. På denne måten sikres det at det er mulighet for å markere hver pakke i en trafikklasse på en bedre måte enn hva som er mulig i en tcpdump-fil.

5.4.1 Persistent lagring av treningsdata



Figur 5. Databasen for lagring av treningsdata

Som vi ser av Figur 5 har databasen struktur som et datavarehus, og den lagrer all informasjon om hver pakke som leses inn. Hvert felt i IP og TCP/UDP protokollen lagres sammen med de første 10 bytene av pakkeinnholdet.

I databasen er hver pakke merket med hva slags trafikktype pakken tilhører. De ulike trafikklassene en pakke kan ha er Normal, DOS, U2R, Probe eller R2L. Merkingen av pakker er en manuell prosess der Darpa datasettets informasjon om de ulike sesjonene legges til grunn for merkingen.

Når merkingen er komplett er treningsgrunnlaget for klassifisering til stede. Fra databasen kan det hentes ut pakker fra alle trafikklasser slik at et ønsket treningssett for den aktuelle klassifikatoren kan genereres.

Det er flere fordeler ved å ha alle treningsdata samlet i en database, i motsetning til å ha dem i tcpdump filer:

- Det gjør det enklere å plukke ut en bestemt pakke for å inspisere dens attributter.
- Det gjør det enklere å finne ut antall angrepspakker, og hvilke pakker som er angrepspakker, for deretter kunne beregne en forvirringsmatrise.
- Det er ikke nødvendig å kjøre en tidkrevende innlesing av treningsdata hver gang klassifikatoren skal startes.

Den største ulempen ved å bruke persistent lagring av treningsdata, er at det kreves en kraftig database som kan takle store datamengder uten at ytelsen faller for mye.

5.5 Naiv Bayesiansk klassifikator

Den Naive Bayesianske klassifikatoren er en relativt enkel form for en maskinlæringsalgoritme. Den har vist seg effektiv når det gjelder hastighet samtidig som treffsikkerheten er rimelig god hvis den blir trent med egnete data. Naiv Bayesiansk klassifikator baserer seg på Bayes Teorem:

$$P(a \mid B) = \frac{P(B \mid a) * P(a)}{P(B)}$$

- Der *P(a)* er sannsynligheten for hypotesen a.
- *P(B)* er sannsynligheten av treningsdata for B.
- *P(a|B)* er sannsynligheten for a gitt B.
- *P(B|a)* er sannsynligheten for B gitt a.

Dette teoremet ligger til grunn for klassifikatoren som brukes senere i oppgaven. Der vil *P(B|a)* angi sannsynligheten for attributtverdien B gitt klassen a.
Hver nettverkspakke vil ha et antall attributter, og den totale sannsynligheten regnes som

$$\sum_{0}^{Antallattributter} \log(P(a \mid B))$$

Der *P(a|B)* er sannsynligheten for hver attributtverdi a gitt klassen *B.* Denne utregningen gjøres for hver klasse av nettverkstrafikk, og den klassen med høyest sannsynlighet velges.

5.5.1 Eksempel på naiv Bayesiansk klassifikator

For å understreke enkelheten i den Naive Bayesianske klassifikatoren gir vi et lett tenkt eksempel på hvordan den virker i praksis.

Vi tenker oss følgende enkle scenario:
Vi skal finne ut om en bil må på verksted. Utfallet kan bli at bilen må på verksted eller at den er ok. Dette måler vi ved hjelp av 3 attributter:
Attributt 1: Slitasje på bremsene, som kan ha stadiene ingen, middels og stor.
Attributt 2: Nivå på motorolje, som kan være full, middels og tom.
Attributt 3: Strøm på batteriet, som kan være fullt, middels og tomt.

Ved å studere biler som vi vet om har måttet på verksted eller er ok, har vi funnet ut at vi har følgende sannsynligheter:

Attributt 1		
	På verksted	Ikke på verksted
Ingen	0,1	0,9
Middels	0,6	0,4
Stor	0,95	0,05

Attributt 2		
	På verksted	Ikke på verksted
Full	0,08	0,92
Middels	0,55	0,45
Tom	0,99	0,01

Attributt 3		
	På verksted	Ikke på verksted
Fullt	0,3	0,7
Middels	0,73	0,27
Tomt	0,992	0,008

Med dette grunnlaget kan vi nå si noe om hvor stor sannsynlighet det er for en bil at den må på verksted. Hvis vi studerer en bil og ser at bilen har stor slitasje på bremsene, motoroljen er full og det er middels med strøm på batteriet vil bruk av den Naive Bayesianske klassifikatoren gi oss følgende:
Sannsynlighet for at bilen må på verksted:

$$\sum_{0}^{Antallattributter} \log(P(a \mid B)) = \log(0{,}95) + \log(0{,}08) + \log(0{,}73) = -2{,}89$$

Sannsynligheten for at bilen ikke skal på verksted er da:

$$\sum_{0}^{Antallattributter} \log(P(a \mid B)) = \log(0{,}05) + \log(0{,}92) + \log(0{,}27) = -4{,}39$$

Disse verdiene beregnes på bakgrunn av formlene gitt i kapittel 5.5. Den naive Bayesianske klassifikatoren sier at man da skal sammenlikne verdiene for de ulike klassene for så å velge den klassen med høyest sannsynlighet. I dette tilfellet betyr det at bilen vi studerte må på verksted.

6 Attributtutvelgelse

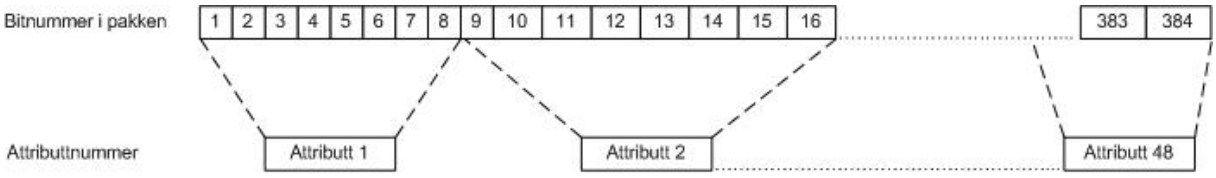
For å trene en Naiv Bayesiansk klassifikator trengs det et datasett. Datasettet er en samling av den typen data en ønsker å klassifisere, fordelt på de ulike alternativene klassifikatoren har og velge mellom. I denne oppgaven er de ulike alternativene de trafikklassene som beskrives i Darpa datasettet (Normal, DOS, Probe, U2R, R2L). For å kunne bruke disse trafikklassene til opptrening av en Naiv Bayesiansk klassifikator må hver av klassene deles opp i grupperes i attributter på en hensiktsmessig måte.

Videre i kapittelet ser vi på hvordan ulike attributtene kan velges ut, og hvordan de ulike måtene å velge attributter på har innvirkning på klassifikatorens treffsikkerhet. Det finnes flere måter å tenke på når vi skal gruppere datasettet i attributter. I denne oppgaven, som omhandler klassifisering av nettverkstrafikk, kan en innfallsvinkel være å se på feltene i IP og TCP/UDP-headeren pluss de 8 første bytene av pakkeinnholdet som attributter. Ser vi på datasettet på denne måten vil vi få 12 attributter for feltene i IP headeren, 15 attributter for feltene i TCP-headeren og 4 attributter for feltene i UDP headeren. I tillegg får vi ett attributt for hver av de 8 første bytene i pakkeinnholdet. Det betyr at vi for en TCP pakke vil ha totalt 35 attributter og en UDP pakke har 24 attributter.
Ser vi nærmere på attributtene i en TCP pakke, når attributtene defineres som over, vil det være svært ulik størrelse på disse. De minste attributtene vil være 1 bit (f.eks. flaggene i TCP headeren), mens de største vil være 32 bits (f.eks. Destination address i IP headeren). Denne ulikheten i attributtstørrelse kan føre til problemer for klassifikatoren. Problemene består i at for små attributter kan føre til at vi mister informasjon om relasjoner i datasettet. I en TCP pakke kan flere av flaggene i TCP-headeren være satt på en gang. Når flere flagg er satt på en gang betyr det at det er relasjon mellom disse feltene. Ved å bruke hvert felt som et eget attributt vil denne relasjonen splittes opp og hvert av disse attributtene blir klassifisert hver for seg. Når et slikt 1-bits attributt klassifiseres kan attributter bare anta to ulike verdier, 0 og 1. Hvis vi derimot hadde hatt ett attributt bestående av 2 av flaggene i TCP-headeren ville det være mulig med fire ulike verdier på attributtet. Ved 4 mulige verdier av attributtet vil det være bedre muligheter for klassifikatoren til å skille de ulike klassene fra hverandre i og med at det i en klasse kan være vanlig at begge flaggene er satt, mens det i en annen kan være vanlig at 1 av flaggene er satt. Ved bruk av 1-bits attributter vil vi miste denne typen sammenhenger mellom attributter.

I motsatt tilfelle har vi de store attributtene på 32 bits. Hvert av disse 32 bits attributtene kan anta over 4 milliarder verdier. For å dekke alle verdiene kreves et datasett som inneholder alle verdiene for hver av klassene, noe som medfører at datasettet må være enormt stort. Hvis ikke alle verdiene er dekket i hver av klassene kan det bety at data som skal klassifiseres blir klassifisert mer eller mindre tilfeldig ut ifra hvilken klasse som dekker verdien i attributtet på pakken som skal klassifiseres. I praksis betyr dette at det bør være en relativt lik fordeling

mellom treningsdata i de ulike klassene som skal klassifiseres, slik at vi unngår høyere hyppighet av en klasse på grunn av at klassen har mer treningsdata enn de andre klassene.
Resultatene som beskrives i kapittel 7.1 og kapittel 7.2 baserer seg på attributtvalg som beskrevet ovenfor.

En annen innfallsvinkel vi kan bruke er å betrakte pakken som en samling av bits. I vårt tilfelle består denne samlingen av 384 bits (48 bytes) Denne samlingen av bits kan da deles opp på ulike måter. NETAD[1] bruker 8 bits i hvert attributt. Disse 8 bitene plukkes ut sekvensielt fra pakken, slik at attributt 1 inneholder bit nummer 1-8 i pakken, attributt nr 2 inneholder bit nummer 9-16 osv (se Figur 6)



Figur 6. Sekvensiell utvelgelse av bits til attributter

Denne metoden å gruppere pakkens bits på gir oss 48 like store attributter. Attributtene vil nå kunne være delt over ett eller flere av headerfeltene som er definert i IP og TCP/UDP headeren. Dette har sine fordeler da vi unngår 1-bits attributter og 32 bits attributter. Det kan imidlertid også ha sine ulemper ved at bits tilhørende headerfelt som logisk sett ikke hører sammen nå vil tilhøre samme attributt, eller at headerfelt som har bits som burde være i samme attributt nå blir splittet på 2 ulike attributter.
Kapittel 7.3 bruker attributter som består av bits plukket sekvensielt fra de 384 første bitene i en pakke for å generere resultater.

Hvilke bits som hører sammen er en definisjonssak. I IP og TCP/UDP protokollene defineres denne sammenhengen som bits tilhørende det samme headerfeltet. Dette gir oss den semantiske sammenhengen mellom bitene på den måte at vi kan si at for eksempel bit nummer 8-15 i en pakke gir oss ”Type of service” (definerer protokollen som IP skal frakte, typisk TCP eller UDP). På denne måten kan vi fortsette å definere den semantiske betydningen av hver enkelt bit. Når vi tenker på denne måten og hvert attributt tilsvarer et av headerfeltene i IP og TCP/UDP kan vi si at bitene i pakken er grupper ut ifra semantisk betydning.
Når vi grupperer i attributter med 8 og 8 bits plukket sekvensielt ut ifra pakken distanserer vi oss fra den semantiske betydningen av hver bit og fordelingene på bits blir mer eller mindre tilfeldig i og med at ett attributt kan inneholde bits fra flere headerfelter. For å unngå denne tilfeldige kombineringen av bits er det mulig å se på hvilke mønstre som går igjen i de 384 første bitene i pakken. Det mønsteret vi ønsker å finne, er om det er noen sammenheng mellom to og to bits i pakken. En sammenheng mellom bits kan tyde på at det er en semantisk

sammenheng mellom bitene fordi bitene settes systematisk til visse kombinasjoner i henhold til protokoll og applikasjon som brukes. Hvis denne sammenhengen er sterk nok bør disse to bitene grupperes i samme attributt.
For å måle sammenhengen baserer vi oss på den akselererte LC algortimen[4] som beskriver hvordan vi kan måle avhengigheten mellom variabler i et datasett. For å finne ut hvordan sammenhengen mellom 2 bits er, måles antall ganger de forskjellige kombinasjonene (00,01,10,11) av bitene forekommer.
For hver kombinasjon av bitpar som er målt regnes avhengigheten ut ved følgende uttrykk:

$$P_{uv}(0,0) * \log \frac{P_{uv}(0,0)}{P_u(0) * P_v(0)} + P_{uv}(0,1) * \log \frac{P_{uv}(0,1)}{P_u(0) * P_v(1)} + P_{uv}(1,0) * \log \frac{P_{uv}(1,0)}{P_u(1) * P_v(0)} + P_{uv}(1,1) * \log \frac{P_{uv}(1,1)}{P_u(1) * P_v(1)}$$

Der

$$P_{uv}(x, y) = \frac{Ant_{xy}}{Totalantall}$$

Hvor Ant_{xy} er antall forekomster av kombinasjonen mellom 2 bits, og *Totalantall* er total antall pakker.

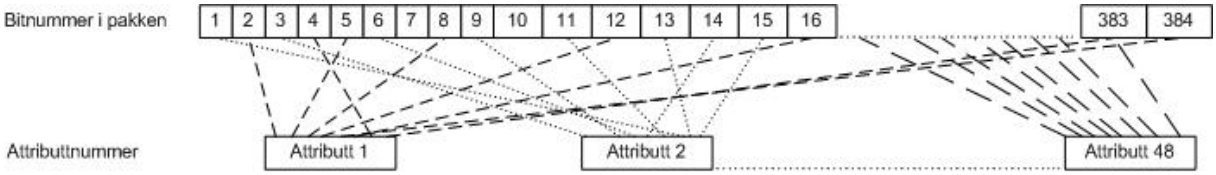
$$P_u(x) = \frac{Ant_x}{Totalantall}$$

Hvor Ant_x er antall ganger x = 1 og *Totalantall* er totalt antall pakker.

$$P_v(y) = \frac{Ant_y}{Totalantall}$$

Hvor Ant_y er total antall ganger den korresponderende biten i bitparet er 1. *Totalantall* er totalt antall pakker.

Den sammenhengen som her er målt betegnes videre som *avhengigheten* mellom 2 bits. Dette er et relativt tall som kun gjelder for det datasettet den er målt for. En endring i datasettet vil medføre at avhengigheten forandres, og det betyr at vi ikke uten videre kan sammenlikne avhengigheter for ulike datasett.
Bruker vi denne måten å gruppere bitene på vil vi få attributter bestående av bits fra alle de 384 bitene, avhengig av hvilke avhenger mest av hverandre. Figur 7 viser et eksempel på dette.



Figur 7. Eksempel på utvelgelse av bits ved bruk av bitavhengighets algoritme

Ser vi litt nærmere på hvordan vi velger bits til attributter i denne oppgaven baserer vi oss på følgende fremgangsmåte:

1. *Opptelling av bitkombinasjoner*:

   For hver pakke i treningsdata:

   For hver bit i pakken:

   Telle opp kombinasjonen av bit og bit+1, og lagre denne i en dictionary

2. *Beregning av avhengigheter:*

   For hver kombinasjon av bit og bit+1:

   Beregne $P_{uv}(0,0)$, $P_{uv}(0,1)$,$P_{uv}(1,0)$,$P_{uv}(1,1)$, $P_u(0)$, $P_u(1)$, $P_v(0)$ og $P_v(1)$ der u = bit og v = bit+1.

   Bruke de beregnede attributtene til å beregne avhengigheten

   $$P_{uv}(0,0)*\log\frac{P_{uv}(0,0)}{P_u(0)*P_v(0)}+P_{uv}(0,1)*\log\frac{P_{uv}(0,1)}{P_u(0)*P_v(1)}+P_{uv}(1,0)*\log\frac{P_{uv}(1,0)}{P_u(1)*P_v(0)}+P_{uv}(1,1)*\log\frac{P_{uv}(1,1)}{P_u(1)*P_v(1)}$$

   Lagre denne avhengigheten for hver kombinasjon av bit og bit+1

3. *Gruppering av bit i attributter:*

   Gå gjennom alle avhengigheter og finn den største avhengigheten blant bitene som ikke er behandlet.

   Marker bitene som har den største avhengigheten som behandlet

   Inntil det er plukket ut 8 bit plukkes det ut bit som har høyest avhengighet til bitene som allerede er plukket ut

Denne fremgangsmåten er en grov oversikt på hva vi gjør for å gruppere 8 og 8 bits ut ifra avhengigheten mellom bitene i pakken. Komplett kode for denne algoritmen finnes i Vedlegg A.

Når vi kjører algoritmen på datasettet vi bruker til trening blir bitene i pakken gruppert på slik som vist i Figur 8. Fra Figur 8 kan det synes som om attributtene består av relativt mange attributter fra ett og samme IP/TCP/UDP headerfelt. Dette er i tilfelle i de fleste attributtene og det kan tyde på at den semantiske betydningen av bitene også gjenspeiler avhengigheten mellom dem. Vi ser også at blant attributtene med laves nummer finnes det mange bits fra felter som kildeadresse, destinasjonsadresse, kildeport og destinasjonsport. Dette kan komme av at dette er felter som er konstante gjennom en dataoverføringssesjon. Kombinasjonen av de nevnte feltene vil da forekomme på alle pakkene i en sesjon mens andre felt ville kunne variere. Resultater oppnådd med attributtene som beskrevet ovenfor finnes i kapittel 7.4.

Byte 1
Bit 1 i Byte 7
Bit 2 i Byte 7
Bit 16 i Destination port
Bit 13 i Destination port
Bit 15 i Destination port
Bit 14 i Destination port
Bit 12 i Destination port
Bit 11 i Destination port

Byte 2
Bit 23 i Source address
Bit 24 i Source address
Bit 21 i Source address
Bit 22 i Source address
Bit 20 i Source address
Bit 19 i Source address
Bit 17 i Source address
Bit 18 i Source address

Byte 3
Bit 4 i Source address
Bit 5 i Source address
Bit 3 i Source address
Bit 2 i Source address
Bit 1 i Source address
Bit 10 i Header Checksum
Bit 11 i Header Checksum
Bit 7 i Header Checksum

Byte 4
Bit 10 i Identification
Bit 11 i Identification
Bit 7 i Identification
Bit 1 i Identification
Bit 8 i TOS
Bit 7 i TOS
Bit 6 i TOS
Bit 5 i TOS

Byte 5
Bit 1 i Header Checksum
Bit 2 i Header Checksum
Bit 3 i TTL
Bit 6 i TTL
Bit 13 i Fragment Offset
Bit 12 i Fragment Offset
Bit 11 i Fragment Offset
Bit 10 i Fragment Offset

Byte 6
Bit 3 i Total Length
Bit 2 i Identification
Bit 6 i Total Length
Bit 4 i TOS
Bit 4 i IHL
Bit 3 i IHL
Bit 2 i IHL
Bit 1 i IHL

Byte 7
Bit 10 i Destination port
Bit 17 i Sequence number
Bit 9 i Destination port
Bit 16 i Source port
Bit 15 i Source port
Bit 14 i Source port
Bit 13 i Source port
Bit 12 i Source port

Byte 8
Bit 11 i Source port
Bit 17 i Ack number
Bit 10 i Source port
Bit 9 i Source port
Bit 8 i Source port
Bit 7 i Source port
Bit 6 i Source port
Bit 5 i Source port

Byte 9
Bit 4 i Source port
Bit 22 i Sequence number
Bit 3 i Source port
Bit 1 i Source port
Bit 5 i Destination address
Bit 4 i Destination address
Bit 3 i Destination address
Bit 2 i Destination address

Byte 10
Bit 1 i Destination address
Bit 4 i Ack number
Bit 32 i Source address
Bit 31 i Source address
Bit 30 i Source address
Bit 27 i Source address
Bit 26 i Source address
Bit 25 i Source address

Byte 11
Bit 5 i Identification
Bit 14 i Identification
Bit 12 i Identification
Bit 11 i Total Length
Bit 2 i Total Length
Bit 1 i TOS
Bit 4 i Version
Bit 3 i Version

Byte 12
Bit 2 i Version
Bit 14 i Total Length
Bit 1 i Version
Bit 14 i Header Checksum
Bit 5 i Header Checksum
Bit 12 i Header Checksum
Bit 3 i Protocol
Bit 2 i TTL

Byte 13
Bit 9 i Fragment Offset
Bit 6 i Protocol
Bit 6 i Fragment Offset
Bit 5 i Fragment Offset
Bit 4 i Fragment Offset
Bit 3 i Fragment Offset
Bit 2 i Fragment Offset
Bit 1 i Fragment Offset

Byte 14
Bit 16 i Source address
Bit 13 i Destination address
Bit 15 i Source address
Bit 14 i Source address
Bit 13 i Source address
Bit 12 i Source address
Bit 11 i Source address
Bit 10 i Source address

Byte 15
Bit 9 i Source address
Bit 2 i Ack number
Bit 8 i Source address
Bit 7 i Source address
Bit 6 i Source address
Bit 3 i Header Checksum
Bit 3 i Flags
Bit 2 i Flags

Byte 16
Bit 1 i Flags
Bit 12 i Checksum
Bit 3 i Identification
Bit 1 i Checksum
Bit 1 i Window
Bit 1 i Destination port
Bit 7 i Destination address
Bit 28 i Source address

Byte 17
Bit 20 i Sequence number
Bit 28 i Sequence number
Bit 29 i Source address
Bit 3 i Ack number
Bit 4 i Byte 7
Bit 3 i Byte 7
Bit 4 i Byte 8
Bit 3 i Byte 8

Byte 18
Bit 4 i TTL
Bit 5 i TTL
Bit 1 i TTL
Bit 8 i Fragment Offset
Bit 16 i Identification
Bit 22 i Ack number
Bit 20 i Ack number
Bit 6 i Destination address

Byte 19
Bit 16 i Header Checksum
Bit 21 i Sequence number
Bit 8 i Protocol
Bit 9 i Destination address
Bit 11 i Destination address
Bit 2 i Source port
Bit 16 i Total Length
Bit 16 i Destination address

Byte 20
Bit 18 i Sequence number
Bit 19 i Sequence number
Bit 5 i Protocol
Bit 13 i Header Checksum
Bit 7 i Protocol
Bit 9 i Header Checksum
Bit 2 i Protocol
Bit 1 i Protocol

Byte 21
Bit 8 i Checksum
Bit 10 i Checksum
Bit 9 i Checksum
Bit 6 i Checksum
Bit 27 i Sequence number
Bit 2 i Data offset
Bit 4 i Data offset
Bit 26 i Sequence number

Byte 22
Bit 8 i Identification
Bit 15 i Identification
Bit 13 i Identification
Bit 9 i Identification
Bit 8 i Header Checksum
Bit 6 i Header Checksum
Bit 4 i Header Checksum
Bit 4 i Protocol

Byte 23
Bit 8 i TTL
Bit 5 i Destination port
Bit 4 i Destination port
Bit 3 i Destination port
Bit 7 i TTL
Bit 7 i Fragment Offset
Bit 15 i Header Checksum
Bit 7 i Destination port

Byte 24
Bit 2 i Destination port
Bit 8 i Destination port
Bit 6 i Destination port
Bit 15 i Destination address
Bit 28 i Ack number
Bit 25 i Sequence number
Bit 5 i Ack number
Bit 30 i Sequence number

Byte 25
Bit 8 i Window
Bit 12 i Window
Bit 9 i Window
Bit 31 i Sequence number
Bit 23 i Sequence number
Bit 6 i Window
Bit 1 i Ack number
Bit 27 i Ack number

Byte 26
Bit 26 i Ack number
Bit 3 i Data offset
Bit 23 i Ack number
Bit 29 i Sequence number
Bit 21 i Ack number
Bit 12 i Destination address
Bit 8 i Destination address
Bit 32 i Sequence number

Byte 27
Bit 4 i Total Length
Bit 5 i Total Length
Bit 1 i Total Length
Bit 3 i TOS
Bit 13 i Total Length
Bit 10 i Total Length
Bit 12 i Total Length
Bit 6 i Identification

Byte 28
Bit 24 i Sequence number
Bit 18 i Ack number
Bit 14 i Ack number
Bit 10 i Destination address
Bit 6 i Ack number
Bit 10 i Window
Bit 1 i Reserved
Bit 9 i Ack number

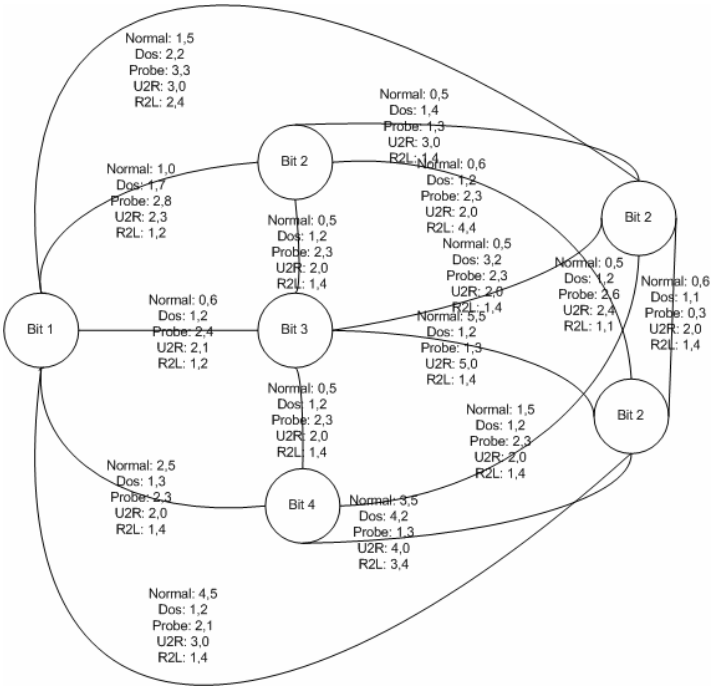
Byte 29	Byte 34	Byte 39	Byte 44
Bit 9 i Total Length	Bit 11 i Checksum	Bit 12 i Ack number	Bit 4 i Window
Bit 15 i Total Length	Bit 2 i Urgent pointer	Bit 13 i Ack number	Bit 1 i Byte 3
Bit 8 i Total Length	Bit 1 i Byte 2	Bit 16 i Ack number	Bit 3 i Window
Bit 7 i Total Length	Bit 3 i Byte 2	Bit 11 i Ack number	Bit 14 i Window
Bit 2 i TOS	Bit 8 i Byte 1	Bit 1 i FIN	Bit 15 i Window
Bit 4 i Identification	Bit 1 i Urgent pointer	Bit 1 i SYN	Bit 16 i Window
Bit 24 i Ack number	Bit 15 i Checksum	Bit 19 i Ack number	Bit 8 i Byte 8
Bit 32 i Ack number	Bit 14 i Checksum	Bit 1 i PSH	Bit 1 i Sequence number
Byte 30	**Byte 35**	**Byte 40**	**Byte 45**
Bit 7 i Checksum	Bit 25 i Ack number	Bit 7 i Ack number	Bit 17 i Destination address
Bit 7 i Byte 1	Bit 1 i RST	Bit 10 i Ack number	Bit 24 i Destination address
Bit 7 i Urgent pointer	Bit 30 i Ack number	Bit 8 i Ack number	Bit 23 i Destination address
Bit 6 i Byte 1	Bit 31 i Ack number	Bit 5 i Reserved	Bit 4 i Sequence number
Bit 8 i Urgent pointer	Bit 1 i Data offset	Bit 4 i Reserved	Bit 5 i Sequence number
Bit 1 i Byte 1	Bit 2 i Reserved	Bit 3 i Reserved	Bit 26 i Destination address
Bit 6 i Urgent pointer	Bit 1 i URG	Bit 1 i ACK	Bit 16 i Sequence number
Bit 14 i Urgent pointer	Bit 29 i Ack number	Bit 6 i Reserved	Bit 32 i Destination address
Byte 31	**Byte 36**	**Byte 41**	**Byte 46**
Bit 5 i Checksum	Bit 2 i Byte 8	Bit 5 i Window	Bit 18 i Destination address
Bit 3 i Byte 1	Bit 5 i Byte 8	Bit 13 i Window	Bit 25 i Destination address
Bit 8 i Byte 2	Bit 1 i Byte 8	Bit 2 i Byte 4	Bit 30 i Destination address
Bit 4 i Byte 1	Bit 8 i Byte 7	Bit 6 i Byte 5	Bit 10 i Sequence number
Bit 13 i Urgent pointer	Bit 7 i Byte 7	Bit 1 i Byte 5	Bit 29 i Destination address
Bit 16 i Urgent pointer	Bit 6 i Byte 7	Bit 6 i Byte 3	Bit 27 i Destination address
Bit 3 i Urgent pointer	Bit 5 i Byte 7	Bit 1 i Byte 4	Bit 9 i Sequence number
Bit 10 i Urgent pointer	Bit 7 i Window	Bit 8 i Byte 4	Bit 15 i Sequence number
Byte 32	**Byte 37**	**Byte 42**	**Byte 47**
Bit 2 i Checksum	Bit 14 i Destination address	Bit 2 i Window	Bit 3 i Sequence number
Bit 7 i Byte 2	Bit 7 i Byte 8	Bit 4 i Byte 3	Bit 13 i Sequence number
Bit 5 i Urgent pointer	Bit 6 i Byte 8	Bit 7 i Byte 4	Bit 12 i Sequence number
Bit 9 i Urgent pointer	Bit 8 i Byte 6	Bit 6 i Byte 4	Bit 8 i Sequence number
Bit 15 i Urgent pointer	Bit 7 i Byte 6	Bit 3 i Byte 4	Bit 11 i Sequence number
Bit 4 i Urgent pointer	Bit 6 i Byte 6	Bit 5 i Byte 3	Bit 14 i Sequence number
Bit 5 i Byte 1	Bit 5 i Byte 6	Bit 2 i Byte 6	Bit 7 i Sequence number
Bit 12 i Urgent pointer	Bit 4 i Byte 6	Bit 5 i Byte 4	Bit 20 i Destination address
Byte 33	**Byte 38**	**Byte 43**	**Byte 48**
Bit 4 i Checksum	Bit 13 i Checksum	Bit 11 i Window	Bit 21 i Destination address
Bit 2 i Byte 1	Bit 16 i Checksum	Bit 3 i Byte 6	Bit 31 i Destination address
Bit 5 i Byte 2	Bit 4 i Byte 5	Bit 1 i Byte 6	Bit 28 i Destination address
Bit 6 i Byte 2	Bit 7 i Byte 5	Bit 4 i Byte 4	Bit 2 i Sequence number
Bit 11 i Urgent pointer	Bit 5 i Byte 5	Bit 2 i Byte 3	Bit 22 i Destination address
Bit 2 i Byte 2	Bit 7 i Byte 3	Bit 3 i Byte 3	Bit 6 i Sequence number
Bit 3 i Checksum	Bit 8 i Byte 3	Bit 2 i Byte 5	Bit 19 i Destination address
Bit 4 i Byte 2	Bit 3 i Byte 5	Bit 8 i Byte 5	Bit 15 i Ack number

Figur 8. Fordeling av bits på attributter ved bitavhengighet på 3 ukers treningsdata

Ved å benytte denne måten å gruppere bits på har vi to problemer. Det ene består i at vi grupperer bits som ikke på noen måte har noen sterk avhengighet til hverandre. Dette er tilfellet i de siste attributtene som blir gruppert. Dermed kan det bety at vi egentlig ikke får noe særlig hjelp i klassifiseringen da disse siste attributtene kan anta nesten alle mulige lovlige verdier, og sannsynligheten vil da være omtrent lik for alle de ulike verdiene. Det neste problemet består i at mengden treningsdata for hver klasse er svært ulikt. Dette fører til at mønstre som opptrer i en klasse med lite treningsdata kan bli undertrykt av mønstre som forekommer i klasser med store mengder treningsdata. Dette gjelder særlig forholdet mellom normaltrafikk og angrepstrafikk. Mengden av treningsdata for normaltrafikk er mye større enn mengden av treningsdata for hver av angrepsklassene, noe som fører til at mønstre som er

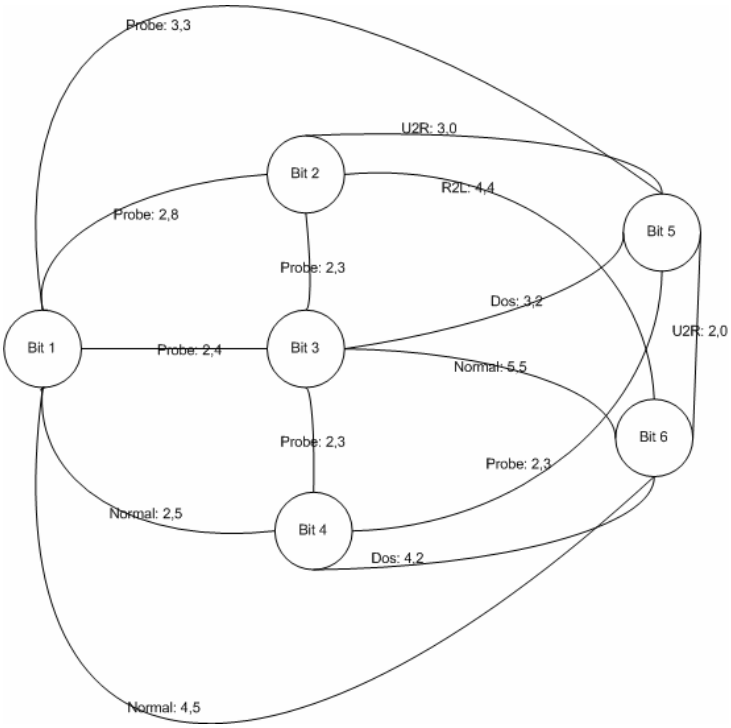
spesielle for en angrepsklasse går tapt i mengden. Dette kan igjen medføre at det blir vanskeligere å detektere angrepstrafikk.
For å løse disse problemene foreslår vi å lage attributter av ulik størrelse, dog med maksimal størrelse på 8 bits. Størrelsen av attributtene skal styres med en terskel som sier at bits skal ha en avhengighet større enn terskelen for å bli gruppert sammen.

For å løse problemet med at viktige mønstre i angrepsklassene kan gå tapt vil det nå beregnes avhengigheter mellom bits i hver klasse, for deretter beregne en endelig bitsammensetning ut ifra de høyeste avhengighetene i hver klasse. Dette er illustrert under.



Figur 9. En fiktiv graf med 5 bits og kanter som har avhengighetsverdier for alle klasser.

Som vi ser av Figur 9 har hver kant i grafen 5 avhengighetsverdier. Det vil til en hver tid være den høyeste avhengighetsverdien mellom to bits som gjelder, som fører til at den endelige grafen blir seende slik ut.



Figur 10. En tenkt graf med de aktuelle avhengighetsverdiene på kantene

Figur 10 viser de høyeste avhengighetsverdiene plassert på hver kant. Ut ifra disse verdiene blir nå bits gruppert. Da det ikke skal være noe bestemt antall bits i hver gruppe, bortsett fra at det skal være maks 8, bruker vi terskelen som er beskrevet tidligere til å si noe om hvor grensen for gruppering skal gå. Hvis vi tar utgangspunkt i Figur 10, og setter en terskel på 3,5 får vi følgende gruppering: Bit3, Bit6, Bit2, Bit4 og Bit1 i et attributt mens Bit 5 blir stående alene i et attributt. Som vi ser av dette vil en økning av terskelen gi flere små attributter, samtidig som en lavere terskel gir flere 8 bits attributter. Derfor gjelder det å sette terskelen slik at vi får færrest mulig 1-bits attributter samtidig som ikke alle attributtene blir 8 bits. Dette betyr at vi får følgende fremgangsmåte for å gruppere bits:

1. *Opptelling av bitkombinasjoner*:

   For hver trafikklasse:

   For hver pakke i treningsdata:

   For hver bit i pakken:

   Telle opp kombinasjonen av bit og bit+1, og lagre denne i en dictionary

2. *Beregning av avhengigheter:*

   For hver trafikklasse:

   For hver kombinasjon av bit og bit+1:

   Beregne $P_{uv}(0,0)$, $P_{uv}(0,1)$,$P_{uv}(1,0)$,$P_{uv}(1,1)$, $P_u(0)$, $P_u(1)$, $P_v(0)$ og $P_v(1)$ der u = bit og v = bit+1.

   Bruke de beregnede attributtene til å beregne avhengigheten

$$P_{uv}(0,0)*\log\frac{P_{uv}(0,0)}{P_u(0)*P_v(0)}+P_{uv}(0,1)*\log\frac{P_{uv}(0,1)}{P_u(0)*P_v(1)}+P_{uv}(1,0)*\log\frac{P_{uv}(1,0)}{P_u(1)*P_v(0)}+P_{uv}(1,1)*\log\frac{P_{uv}(1,1)}{P_u(1)*P_v(1)}$$

   Lagre denne avhengigheten for hver kombinasjon av bit og bit+1

3. *Gruppering av bit i attributter:*

   Gå gjennom alle avhengigheter og finn den største avhengigheten uavhengig av trafikklasse blant bitene som ikke er behandlet. Finnes det ikke avhengigheter som er større enn terskel vil bitene settes i 1-bits attributter

   Marker bitene som har den største avhengigheten som behandlet

   Så lenge det finnes bit med avhengighet til et av bitene som er plukket ut med større avhengighet enn terskelen som er satt, blir biten lagt til, inntil attributtet har maks 8 bits.

Denne fremgangsmåten skisserer grovt hvordan vi grupperer for å danne attributter med variabel lengde som består av bits som avhenger av hverandre. Komplett kode for denne algoritmen finnes i Vedlegg A.

Bruker vi fremgangsmåten beskrevet over på treningsdatasettet får vi følgende fordeling på attributter:

Attributt 1
Bit 5 i Destination address
Bit 12 i Destination address
Bit 4 i Destination address
Bit 3 i Destination address
Bit 2 i Destination address
Bit 1 i Destination address
Bit 32 i Source address
Bit 31 i Source address

Attributt 2
Bit 13 i Destination port
Bit 9 i Window
Bit 11 i Destination port
Bit 10 i Destination port
Bit 9 i Destination port
Bit 14 i Source port
Bit 13 i Source port
Bit 12 i Source port

Attributt 3
Bit 20 i Sequence number
Bit 1 i Reserved
Bit 16 i Source port
Bit 15 i Source port
Bit 11 i Source port
Bit 10 i Source port
Bit 9 i Source port
Bit 8 i Source port

Attributt 4
Bit 7 i Source port
Bit 11 i Checksum
Bit 6 i Source port
Bit 5 i Source port
Bit 4 i Source port
Bit 1 i Source port
Bit 30 i Source address
Bit 27 i Source address

Attributt 5
Bit 3 i Source port
Bit 27 i Ack number
Bit 23 i Source address
Bit 21 i Source address
Bit 20 i Source address
Bit 19 i Source address
Bit 17 i Source address
Bit 16 i Source address

Attributt 6
Bit 15 i Source address
Bit 7 i Byte 6
Bit 14 i Source address
Bit 13 i Source address
Bit 12 i Source address
Bit 11 i Source address
Bit 10 i Source address
Bit 9 i Source address

Attributt 7
Bit 26 i Source address
Bit 1 i Byte 3
Bit 25 i Source address
Bit 8 i Source address
Bit 7 i Source address
Bit 6 i Source address
Bit 4 i Source address
Bit 3 i Source address

Attributt 8
Bit 2 i Source address
Bit 24 i Sequence number
Bit 1 i Source address
Bit 10 i Header Checksum
Bit 7 i Header Checksum
Bit 3 i Header Checksum
Bit 1 i Header Checksum
Bit 3 i TTL

Attributt 9
Bit 13 i Fragment Offset
Bit 7 i Byte 5
Bit 12 i Fragment Offset
Bit 11 i Fragment Offset
Bit 10 i Fragment Offset
Bit 9 i Fragment Offset
Bit 5 i Fragment Offset
Bit 4 i Fragment Offset

Attributt 10
Bit 3 i Fragment Offset
Bit 26 i Ack number
Bit 2 i Fragment Offset
Bit 1 i Fragment Offset
Bit 3 i Flags
Bit 2 i Flags
Bit 1 i Flags
Bit 10 i Identification

Attributt 11
Bit 8 i TOS
Bit 17 i Sequence number
Bit 7 i TOS
Bit 6 i TOS
Bit 5 i TOS
Bit 4 i TOS
Bit 4 i IHL
Bit 3 i IHL

Attributt 12
Bit 7 i Identification
Bit 6 i Ack number
Bit 3 i Identification
Bit 1 i Identification
Bit 3 i Total Length
Bit 2 i IHL
Bit 1 i IHL
Bit 4 i Version

Attributt 13
Bit 3 i Version
Bit 3 i Byte 6
Bit 2 i Version
Bit 1 i Version
Bit 1 i Checksum
Bit 1 i Window
Bit 8 i Byte 6
Bit 1 i Byte 5

Attributt 14
Bit 26 i Sequence number
Bit 24 i Ack number
Bit 5 i Protocol
Bit 4 i TTL
Bit 1 i TTL
Bit 8 i Fragment Offset
Bit 2 i Protocol
Bit 6 i Header Checksum

Attributt 15
Bit 16 i Identification
Bit 32 i Sequence number
Bit 5 i Identification
Bit 16 i Total Length
Bit 11 i Total Length
Bit 2 i Total Length
Bit 1 i TOS
Bit 8 i Header Checksum

Attributt 16
Bit 28 i Source address
Bit 2 i Byte 6
Bit 12 i Destination port
Bit 7 i Destination address
Bit 16 i Destination port
Bit 5 i Byte 6
Bit 1 i Destination port
Bit 6 i Byte 6

Attributt 17
Bit 20 i Ack number
Bit 15 i Checksum
Bit 13 i Total Length
Bit 29 i Sequence number
Bit 4 i Total Length
Bit 1 i Total Length
Bit 3 i TOS
Bit 6 i Identification

Attributt 18
Bit 12 i Total Length
Bit 6 i Destination address
Bit 10 i Total Length
Bit 32 i Ack number
Bit 4 i Protocol
Bit 27 i Sequence number
Bit 30 i Sequence number
Bit 25 i Sequence number

Attributt 19
Bit 29 i Source address
Bit 8 i Byte 5
Bit 8 i Checksum
Bit 5 i Byte 5
Bit 3 i Byte 5
Bit 4 i Byte 5
Bit 9 i Checksum
Bit 8 i Window

Attributt 20
Bit 1 i Byte 7
Bit 5 i Byte 8
Bit 1 i Byte 8
Bit 2 i Byte 8
Bit 7 i Byte 7
Bit 8 i Byte 7
Bit 3 i Byte 8
Bit 6 i Byte 7

Attributt 21
Bit 9 i Destination address
Bit 3 i Window
Bit 4 i Window
Bit 31 i Sequence number
Bit 1 i Ack number
Bit 13 i Header Checksum
Bit 9 i Header Checksum
Bit 2 i Window

Attributt 22
Bit 6 i Checksum
Bit 2 i Byte 5
Bit 4 i Byte 6
Bit 7 i Checksum
Bit 5 i Checksum
Bit 2 i Checksum
Bit 4 i Checksum
Bit 3 i Checksum

Attributt 23
Bit 1 i Byte 4
Bit 3 i Byte 7
Bit 8 i Destination address
Bit 2 i Byte 4
Bit 4 i Byte 4
Bit 1 i Byte 6
Bit 6 i Byte 4
Bit 8 i Byte **4**

Attributt 24
Bit 5 i Ack number
Bit 29 i Ack number
Bit 23 i Sequence number
Bit 2 i TTL
Bit 6 i Fragment Offset
Bit 8 i Identification
Bit 5 i Header Checksum
Bit 3 i Protocol

Attributt 25
Bit 16 i Destination address
Bit 4 i Ack number
Bit 15 i Destination address
Bit 18 i Sequence number
Bit 13 i Checksum
Bit 25 i Ack number
Bit 30 i Ack number
Bit 31 i Ack number

Attributt 26
Bit 10 i Destination address
Bit 13 i Destination address
Bit 16 i Header Checksum
Bit 8 i Protocol
Bit 23 i Ack number
Bit 7 i Protocol
Bit 1 i Protocol
Bit 7 i TTL

Attributt 27
Bit 5 i Byte 7
Bit 8 i Byte 8
Bit 6 i Byte 8
Bit 7 i Byte 8
Bit 2 i Source port
Bit 7 i Byte 3
Bit 5 i Byte 3
Bit 3 i Byte 4

Attributt 28
Bit 7 i Fragment Offset
Bit 1 i Data offset
Bit 8 i TTL
Bit 15 i Header Checksum
Bit 4 i Header Checksum
Bit 13 i Identification
Bit 9 i Identification
Bit 2 i Destination port

Attributt 29
Bit 6 i Window
Bit 12 i Checksum
Bit 2 i Reserved
Bit 7 i Window
Bit 5 i Window
Bit 4 i Byte 7
Bit 14 i Checksum
Bit 6 i Reserved

Attributt 30
Bit 15 i Total Length
Bit 14 i Identification
Bit 12 i Identification
Bit 9 i Total Length
Bit 7 i Total Length
Bit 2 i TOS
Bit 8 i Total Length
Bit 4 i Identification

Attributt 31
Bit 18 i Ack number
Bit 22 i Ack number
Bit 15 i Ack number
Bit 12 i Ack number
Bit 8 i Ack number
Bit 10 i Ack number
Bit 14 i Destination address
Bit 28 i Sequence number

Attributt 32
Bit 5 i Total Length
Bit 14 i Total Length
Bit 5 i TTL
Bit 15 i Identification
Bit 4 i Data offset
Bit 16 i Ack number
Bit 2 i Data offset
Bit 11 i Ack number

Attributt 33
Bit 17 i Destination address
Bit 22 i Sequence number
Bit 21 i Sequence number
Bit 18 i Destination address
Bit 26 i Destination address
Bit 3 i Sequence number
Bit 21 i Destination address
Bit 7 i Sequence number

Attributt 34
Bit 11 i Window
Bit 4 i Byte 8
Bit 15 i Window
Bit 16 i Checksum
Bit 14 i Window
Bit 16 i Window
Bit 13 i Window
Bit 12 i Window

Attributt 35
Bit 9 i Ack number
Bit 28 i Ack number
Bit 19 i Sequence number
Bit 7 i Ack number
Bit 21 i Ack number
Bit 3 i Data offset
Bit 2 i Sequence number
Bit 2 i Ack number

Attributt 36
Bit 1 i Sequence number
Bit 3 i Ack number
Bit 20 i Destination address
Bit 6 i Sequence number
Bit 16 i Sequence number
Bit 4 i Sequence number
Bit 12 i Sequence number
Bit 28 i Destination address

Attributt 37
Bit 5 i Reserved
Bit 10 i Checksum
Bit 4 i Reserved
Bit 19 i Ack number
Bit 1 i URG
Bit 3 i Reserved
Bit 1 i FIN
Bit 1 i PSH

Attributt 38
Bit 4 i Destination port
Bit 5 i Destination port
Bit 7 i Destination port
Bit 3 i Destination port
Bit 6 i Destination port
Bit 8 i Destination port
Bit 19 i Destination address
Bit 22 i Destination address

Attributt 39
Bit 12 i Header Checksum
Bit 14 i Header Checksum
Bit 6 i Protocol

Attributt 40
Bit 3 i Byte 2
Bit 7 i Byte 2
Bit 1 i Byte 2
Bit 2 i Byte 2
Bit 4 i Byte 2
Bit 6 i Byte 2
Bit 5 i Byte 1
Bit 8 i Byte 1

Attributt 41
Bit 13 i Ack number
Bit 14 i Ack number
Bit 14 i Sequence number
Bit 13 i Sequence number
Bit 29 i Destination address
Bit 8 i Sequence number
Bit 31 i Destination address
Bit 15 i Sequence number

Attributt 42
Bit 10 i Urgent pointer
Bit 3 i Byte 1
Bit 2 i Byte 1
Bit 4 i Byte 1
Bit 1 i Urgent pointer
Bit 1 i Byte 1
Bit 2 i Urgent pointer
Bit 3 i Urgent pointer

Attributt 43
Bit 1 i RST
Bit 1 i SYN
Bit 30 i Destination address
Bit 9 i Sequence number
Bit 23 i Destination address
Bit 5 i Sequence number
Bit 24 i Destination address
Bit 32 i Destination address

Attributt 44
Bit 4 i Urgent pointer
Bit 9 i Urgent pointer
Bit 6 i Urgent pointer
Bit 7 i Urgent pointer
Bit 5 i Urgent pointer
Bit 8 i Urgent pointer
Bit 11 i Urgent pointer
Bit 13 i Urgent pointer

Attributt 45
Bit 11 i Sequence number
Bit 10 i Window
Bit 1 i ACK
Bit 25 i Destination address
Bit 10 i Sequence number
Bit 27 i Destination address

Attributt 46
Bit 3 i Byte 3
Bit 5 i Byte 4
Bit 6 i Byte 3
Bit 7 i Byte 4
Bit 8 i Byte 3
Bit 4 i Byte 3
Bit 2 i Byte 3

Attributt 47
Bit 15 i Urgent pointer

Attributt 48
Bit 12 i Urgent pointer

Attributt 49
Bit 5 i Byte 2

Attributt 50
Bit 6 i Byte 1

Attributt 51
Bit 7 i Byte 1

Attributt 52
Bit 14 i Urgent pointer

Attributt 53
Bit 16 i Urgent pointer

Attributt 54
Bit 14 i Destination port

Attributt 55
Bit 6 i Byte 5

Attributt 56
Bit 15 i Destination port

Attributt 57
Bit 8 i Byte 2

Attributt 58
Bit 11 i Destination address

Attributt 59
Bit 2 i Header Checksum

Attributt 60 Bit 6 i TTL	**Attributt 63** Bit 11 i Header Checksum	**Attributt 66** Bit 24 i Source address	**Attributt 69** Bit 5 i Source address
Attributt 61 Bit 2 i Identification	**Attributt 64** Bit 11 i Identification	**Attributt 67** Bit 22 i Source address	**Attributt 70** Bit 2 i Byte 7
Attributt 62 Bit 6 i Total Length	**Attributt 65** Bit 17 i Ack number	**Attributt 68** Bit 18 i Source address	

Figur 11. Fordeling av bits på attributter med variabel attributtlengde

Ser vi på Figur 11 ser vi at antall attributter er betraktelig økt i forhold til sammensetningen av attributter vist i Figur 8. Denne økningen finner vi først og fremst igjen som 1-bits attributter. Disse 1-bits attributtene består av bit som er svært uavhengige til alle andre bits i pakken. Det betyr igjen at i forholdet til andre bits vil disse skifte verdi relativt ofte. Antallet av disse 1-bits attributtene kan reguleres ved å justere terskelen. En høy terskel gir mange små attributter, mens en terskel på 0 gir attributter på 8 bits slik som i Figur 8. Resultater for klassifisering med attributtene beskrevet i Figur 11 finnes i kapittel 7.5.

7 Resultater

7.1 Feltbasert klassifikasjon på ufiltrert nettverkstrafikk

Metoden baserer seg på å bruke feltene i pakkehodene samt 8 byte av pakkeinnholdet som attributter. Det blir klassifisert på et komplett datasett uten noen tilpasninger.
Klassifikatoren i denne metoden bruker nettverkstrafikk som ikke blir behandlet på noen måte før vi benytter den. Dette betyr at det er fra 500 000 til 800 000 pakker å ta hensyn til for hver dag med trening. Det er bare en ørliten del av dette antallet som er angrepstrafikk, noe som gjør at det er mye større, og bedre treningsgrunnlag for normaltrafikk enn for angrepstrafikk.

Tabell 1. Fordeling av pakker i klasser på 3 ukers ufiltrert nettverkstrafikk

Normal	Dos	U2R	Probe	R2L
7938802	405019	6246	20954	1944

Den første klassifiseringen er gjort på et utvalg av treningsdata, med den hensikt å sjekke at klassifikatoren er implementert på en korrekt måte. For at en pakke skal bli klassifisert som angrep skal den ikke bare ha høyest sannsynlighet, den skal ha 10 ganger høyere sannsynlighet for at det er angrep enn for at det er normaltrafikk. Ved å justere denne faktoren opp eller ned kan vi påvirke hva klassifikatoren skal gjøre når det er relativt lik sannsynlighet for 2 klasser. Settes faktoren til 1 betyr det at klassen med høyeste sannsynlighet velges uansett om sannsynligheten for en annen klasse er tilnærmet lik. Ved å sette faktoren til 10 oppnår vi da, i den grad det er mulig, sikre klassifiseringer av angrepstrafikk.

Tabell 2. Fordeling av pakker i testdata

Normal	Dos	U2R	Probe	R2L
558437		463		

Tabell 3. Resultater fra klassifiseringen

Normal	Dos	U2R	Probe	R2L
558406		461	33	

Tabell 4. Resultater fra klassifisering kun på de 8 første bytene i pakkeinnholdet

Normal	Dos	U2R	Probe	R2L
558060	341	462	122	5

Den neste klassifiseringen med denne klassifikatoren er gjort på et utvalg av treningsdata som inneholder flere typer angrepstrafikk. Vi beholder fortsatt en faktor på 10, som sier at det skal være 10 ganger høyere sannsynlighet for en angrepsklasse enn for normalklassen hvis en pakke skal klassifiseres som angrep.

Tabell 5. Fordeling av pakker i testdata

Normal	Dos	U2R	Probe	R2L
419257	119165	2215	7991	472

Tabell 6. Resultater for klassifiseringen

Normal	Dos	U2R	Probe	R2L
543469		2209	1131	1691

Som vi ser av resultatene i klassifiseringen er ikke resultatene imponerende. Vi ser at Dos angrep ikke detekteres i det hele tatt, samtidig som Probe detekteres dårlig. U2R og R2L angrep går, i motsetning til Dos og Probe mer på innholdet i pakken. Vi ser at U2R og R2L detekteres, selv om vi ut fra disse resultatene ikke kan være sikre på at pakkene som detekteres som angrep virkelig er angrepspakker.
Det kan være nærliggende å tro at grunnen til dårlig treff på klassifiseringen skyldes at det skal være 10 ganger så stor sannsynlighet for angrep som normaltrafikk. Ved å senke denne faktoren slik at klassifikatoren bare velger trafikklassen med høyest sannsynlighet blir det følgende resultater.

Tabell 7. Resultater fra klassifisering med sannsynlighetsfaktor 1

Normal	Dos	U2R	Probe	R2L
229387	101963	34547	14884	167619

Som vi ser endrer resultatene seg drastisk ved å senke faktoren, men det er fortsatt langt i fra gode resultater.

7.2 Feltbasert klassifikasjon på filtrert nettverkstrafikk

De følgende resultatene er generert med samme attributter som i kapittel 7.1. Forskjellen er at det her benyttes en 2 trinns klassifisering, der trinn 1 er et filter som skal begrense mengden nettverkstrafikk mens trinn 2 bruker en Naiv Bayesiansk klassifikator.

Filtreringen som brukes her tar utgangspunkt i NETAD[1] sin filtrering og filteret har følgende egenskaper:

- All ikke IP trafikk (ARP etc.) filtreres bort, noe som reduserer sjansen for falske alarmer.
- All utgående trafikk filtreres bort. Dette kan føre til at uregelmessigheter på svar fra serverne går tapt.
- All TCP trafikk som starter med en SYN-ACK pakke, noe som indikerer at pakken er initiert lokalt filtreres bort. Normalt initieres angrep utenifra mot en local server.
- UDP trafikk til porter >1023 filtreres bort. Trafikk til høyere UDP porter er ofte DNS trafikk o.l.
- TCP trafikk som starter etter at de første 100 bytes er overført (iflg. sekvensnummeret) filtreres bort.
- Pakker til en adresse/port/protokoll kombinasjon der det har kommet mer enn 16 pakker i løpet av 60 sekunder filtreres bort. Dette vil i stor grad være trafikk av typen DOS (angrep)

Filteret kjøres på pakkestrømmen både ved trening og klassifisering av pakker. Filteret fører til at antall pakker som skal behandles hver dag reduseres med opp til 75-80%, noe som også gir en betraktelig raskere klassifisering.

Resultatene som beskrives i avsnittet resultater baserer seg på at klassifikatoren trenes opp med enten 2 eller 3 ukers treningsdata. Pakkefordelingen i trafikklasser på disse periodene finnes i tabellene under.

Tabell 8. Fordeling av pakker i trafikklasser filtrerte treningsdata fra uke 1 og 2

Normal	Dos	U2R	Probe	R2L
1794705	11405	71	655	388

Tabell 9. Fordeling av pakker i trafikklasser ved filtrerte treningsdata fra uke 1-3

Normal	Dos	U2R	Probe	R2L
2108517	242492	181	11007	417

Den første klassifiseringen med den feltbaserte klassifikatoren på filtrerte data ble utført i den hensikt å sjekke implementasjonen av klassifikatoren. Denne klassifiseringen baserer seg på treningsdata fra uke 1 og 2.

Tabell 10. Fordeling av pakker i testdata

Normal	Dos	U2R	Probe	R2L
158559		35		

Tabell 11. Resultater for klassifiseringen

Normal	Dos	U2R	Probe	R2L
158551		43		

I den neste klassifiseringen utvider vi forrige klassifisering med flere angrepstyper. Testdataene er fremdeles et utvalg av treningsdataene.

Tabell 12. Fordeling av pakker i testdata

Normal	Dos	U2R	Probe	R2L
246436			387	

Tabell 13. Resultater av klassifiseringen

Normal	Dos	U2R	Probe	R2L
246318			503	2

Foran denne klassifiseringen ble treningssettet utvidet til 3 ukers datatrafikk for å teste effekten av dette. Testdata er et utvalg av treningsdata og det må fortsatt være 10 ganger så stor sannsynlighet for angrep enn for normaltrafikk for at en pakke skal klassifiseres som angrep.

Tabell 14. Fordeling av pakker i testdata

Normal	Dos	U2R	Probe	R2L
87668	26275	89	9909	19

Tabell 15. Resultater av klassifiseringen

Normal	Dos	U2R	Probe	R2L
126455		87	3459	54

7.3 Klassifisering på attributter med sekvensielt plukkede bits

Resultatene som presenteres i dette kapittelet baserer seg på attributter med lik lengde bestående av 8 bits som er satt sammen ved å plukke bits sekvensielt fra pakken. Det vil med andre ord si at attributt 1 består av bit1-8, attributt 2 av bit 8-15 osv.

Tabell 16. Fordeling av pakker i 3 ukers treningsdata

Normal	Dos	U2R	Probe	R2L
2108517	242492	181	11007	417

Den første klassifiseringen som ble utført var på et utvalg av treningsdataene med den hensikt å teste klassifikatoren.

Tabell 17. Resultater av klassifisering med fast attributtlengde og sekvensielt plukkede bits på et utvalg av treningsdata (onsdag uke 3).

	Normal	Dos	U2R	Probe	R2L	Resultat
Normal	86002	241		1130		98,43%
Dos		26275				100%
U2R			89			100%
Probe				9909		100%
R2L					19	100 %

Som vi ser av disse resultatene ser det ut som alle angrepspakker detekteres. Det forekommer dog en del feilklassifiseringer av normaltrafikk. Dette vil oppleves som falske alarmer. Kjøres klassifikatoren på et datasett som ***ikke*** er et utvalg av treningsdata får vi følgende resultater.

Tabell 18. Resultater av klassifisering med fast attributtlengde og sekvensielt plukkede bits på et utvalg av treningsdata (onsdag uke 4).

	Normal	Dos	U2R	Probe	R2L	Resultat
Normal	56050	34		720		98,67 %
Dos	2	320				99,37 %
U2R						
Probe	627			506		44,66 %
R2L						

Tabell 19. Resultater av klassifisering med fast attributtlengde og sekvensielt plukkede bits på et utvalg av treningsdata (tirsdag uke 5).

	Normal	Dos	U2R	Probe	R2L	Resultat
Normal	103444	71		964		99,00 %
Dos	2	320				99,37 %
U2R	44					0 %
Probe	2813					0 %
R2L						

Tabell 20. Resultater av klassifisering med fast attributtlengde og sekvensielt plukkede bits på et utvalg av treningsdata (mandag uke 6).

	Normal	Dos	U2R	Probe	R2L	Resultat
Normal	104756	39		1264		98,77 %
Dos	800					0%
U2R						
Probe	1			10		90,1 %
R2L	3	8				0 %

Tabell 21. Resultater av klassifisering med fast attributtlengde og sekvensielt plukkede bits på et utvalg av treningsdata (mandag uke 7).

	Normal	Dos	U2R	Probe	R2L	Resultat
Normal	220220	90		1021		99,49 %
Dos	6	1		2		11,11 %
U2R						
Probe	13	4		2		10,53 %
R2L	11					0 %

7.4 Klassifisering på attributter med fast lengde gruppert med bitavhengighetsalgoritmen

Resultatene som presenteres i dette kapittelet baserer seg på attributter med lik lengde bestående av 8 bits som er satt sammen ved å bergegne avhengigheten mellom bitene.

Tabell 22. Fordeling av pakker i 3 ukers treningsdata.

Normal	Dos	U2R	Probe	R2L
2108517	242492	181	11007	417

Den første klassifiseringen som ble utført var på et utvalg av treningsdataene med den hensikt å teste implementasjonen av klassifikatoren.

Tabell 23. Resultater av klassifisering med fast attributtlengde med gjensidig avhengige bits på et utvalg av treningsdata (mandag uke 1).

	Normal	Dos	U2R	Probe	R2L	Resultat
Normal	164690	19		256		99,8%
Dos						
U2R			35			100%
Probe						
R2L						

Utvides dette utvalget til å omfatte flere angrepspakker for å teste klassifiseringen i alle trafikklasser, får vi disse resultatene.

Tabell 24. Resultater av klassifisering med fast attributtlengde med gjensidig avhengige bits på et utvalg av treningsdata (onsdag uke 3).

	Normal	Dos	U2R	Probe	R2L	Resultat
Normal	86820	167		386		99,36%
Dos		26275				100%
U2R			89			100%
Probe				9909		100%
R2L	1				18	94,74%

Som vi ser av disse resultatene ser det ut som alle angrepspakker detekteres, såfremt som 1 R2L pakke. Det forekommer dog en del feilklassifiseringer av normaltrafikk. Dette vil oppleves som falske alarmer. Kjøres klassifikatoren på et datasett som ***ikke*** er et utvalg av treningsdata får vi følgende resultater.

Tabell 25. Resultater av klassifisering med fast attributtlengde med gjensidig avhengige bits (onsdag uke 4).

	Normal	**Dos**	**U2R**	**Probe**	**R2L**	**Resultat**
Normal	56169	24		352		99,34%
Dos	21	301				93,48%
U2R						
Probe	1030			103		9%
R2L						

Tabell 26. Resultater av klassifisering med fast attributtlengde med gjensidig avhengige bits (tirsdag uke 5).

	Normal	**Dos**	**U2R**	**Probe**	**R2L**	**Resultat**
Normal	83144	48		309		99,57 %
Dos	21	302				93,50 %
U2R			44			100 %
Probe	2811	2				0 %
R2L						

Tabell 27. Resultater av klassifisering med fast attributtlengde med gjensidig avhengige bits (mandag uke 6).

	Normal	**Dos**	**U2R**	**Probe**	**R2L**	**Resultat**
Normal	93745	21		392		99,56%
Dos	601	199				24,88 %
U2R						
Probe	11					0 %
R2L	4				7	63,6%

Tabell 28. Resultater av klassifisering med fast attributtlengde med gjensidig avhengige bits (mandag uke 7).

	Normal	Dos	U2R	Probe	R2L	Resultat
Normal	219823	52		456		99,77%
Dos	6	3				33,33%
U2R						
Probe	13					0%
R2L	11					0%

7.5 Klassifisering på attributter med variabel lengde gruppert med bitavhengighetsalgoritmen

Resultatene som presenteres her er et resultat av klassifisering på attributter bestående av gjensidig avhengige bits, men med variabel lengde.
For å kunne sammenlikne metodene blir klassifiseringene for denne metoden utført med samme data som for klassifisering på attributter med fast lengde.

Tabell 29. Fordeling av pakker i 3 ukers treningsdata.

Normal	Dos	U2R	Probe	R2L
2108517	242492	181	11007	417

Den første klassifiseringen som ble utført var på et utvalg av treningsdataene med den hensikt å teste implementasjonen av klassifikatoren.

Tabell 30. Resultater av klassifisering med variabel attributtlengde med gjensidig avhengige bits på et utvalg av treningsdata (mandag uke 1).

	Normal	Dos	U2R	Probe	R2L	Resultat
Normal	164907	16		42		99,96 %
Dos						
U2R			35			100 %
Probe						
R2L						

Utvider vi utvalget slik at vi får pakker fra alle trafikklassene, får vi disse resultatene:

Tabell 31. Resultater av klassifisering med variabel attributtlengde med gjensidig avhengige bits på et utvalg av treningsdata (onsdag uke 3).

	Normal	Dos	U2R	Probe	R2L	Resultat
Normal	87077	127		169		99,66 %
Dos		26275				100 %
U2R			89			100 %
Probe				9909		100 %
R2L					19	100 %

Som vi ser av disse resultatene detekteres alle angrepspakkene. Det forekommer dog en del feilklassifiseringer av normaltrafikk, selv om det er mindre enn i de foregående sammenliknbare resultatene. Disse feilklassifiseringene vil oppleves som falske alarmer som

vi i størst mulig grad bør prøve å begrense. Kjøres klassifikatoren på et datasett som ***ikke*** er et utvalg av treningsdata får vi følgende resultater.

Tabell 32. Resultater av klassifisering med variabel attributtlengde med gjensidig avhengige bits på et utvalg av treningsdata (onsdag uke 4).

	Normal	**Dos**	**U2R**	**Probe**	**R2L**	**Resultat**
Normal	56678	28		98		99,77 %
Dos	2	320				99,37 %
U2R						
Probe	373			760		67,1 %
R2L						

Tabell 33. Resultater av klassifisering med variabel attributtlengde med gjensidig avhengige bits på et utvalg av treningsdata (tirsdag uke 5).

	Normal	**Dos**	**U2R**	**Probe**	**R2L**	**Resultat**
Normal	104381	41		57		99,9 %
Dos	2	320				99,37 %
U2R	44					0 %
Probe	2636	172		5		0,18 %
R2L						

Tabell 34. Resultater av klassifisering med variabel attributtlengde med gjensidig avhengige bits på et utvalg av treningsdata (mandag uke 6).

	Normal	**Dos**	**U2R**	**Probe**	**R2L**	**Resultat**
Normal	105900	20		139		99,85 %
Dos	435	365				45,6 %
U2R						
Probe	8			3		27,3 %
R2L	11					0 %

Tabell 35. Resultater av klassifisering med variabel attributtlengde med gjensidig avhengige bits på et utvalg av treningsdata (mandag uke 7).

	Normal	Dos	U2R	Probe	R2L	Resultat
Normal	221165	52		114		99,92 %
Dos	7	2				22,22 %
U2R						
Probe	14			4		22,22 %
R2L	11					0 %

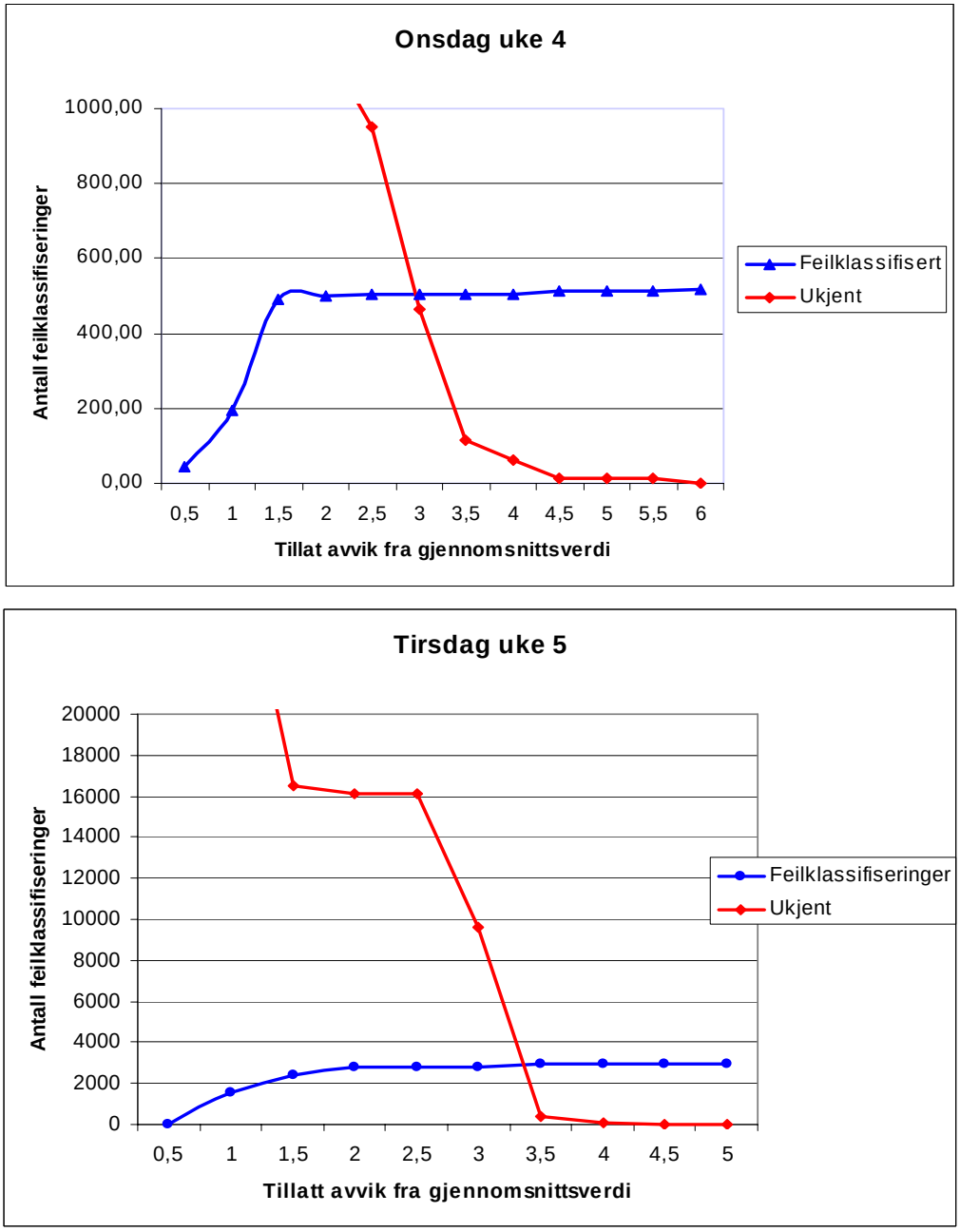
Hvis vi ser litt på resultatene i de foregående tabellene ser vi at det er høy treffprosent på normaltrafikk. Det er imidlertid slik at pakker som ikke kan klassifiseres som angrep blir satt i kategorien normal. Dette er ikke hensiktsmessig da det kan føre til at pakker feilaktig klassifiseres som normaltrafikk. Ved å innføre en klasse ukjent oppstår muligheten til å plassere avvikende pakker i denne kategorien. I og med at vi utgangspunktet sier at pakken tilhører klassen som får utregnet den høyeste sannsynligheten og at ukjent klassen ikke deltar i denne beregningen legges det andre forhold til grunn for at pakker skal klassifiseres som ukjente. Følgende kriterier legges til grunn:

- Pakker som har et stort avvik fra gjennomsnittet til sannsynlighetsverdien i normalklassen klassifiseres som ukjent. Dette avviket kan justeres, men i resultatene som følger skal sannsynligheten for normaltrafikk ligge høyere enn gjennomsnittsverdien for normaltrafikk + 20 %, som tilsvarer en faktor på 1,2 ganger gjennomsnittsverdien.
- Angrepsklassene følger samme modell som normaltrafikk, men da det er normaltrafikken som er den viktigste tillates det en margin på gjennomsnittsverdi + 30 % for angrepstrafikk, som tilsvarer en faktor på 1,3 ganger gjennomsnittsverdien.

Tabell 36. Resultater av klassifisering med variabel attributtlengde med gjensidig avhengige bits på et utvalg av treningsdata når vi innfører klassen ukjent.

Onsdag uke 4							
	Normal	**Dos**	**U2R**	**Probe**	**R2L**	**Ukjent**	**Resultat**
Normal	54917	33		97		1498	97,1 %
Dos		316				6	98,1 %
U2R							
Probe	361			485		287	42,8 %
R2L							
Tirsdag uke 5							
	Normal	**Dos**	**U2R**	**Probe**	**R2L**	**Ukjent**	**Resultat**
Normal	87185	42		66		16867	83,7 %
Dos		315				7	97,8 %
U2R	34					10	0 %
Probe	1494	80				1239	0%
R2L							
Mandag uke 6							
	Normal	**Dos**	**U2R**	**Probe**	**R2L**	**Ukjent**	**Resultat**
Normal	84968	20		154		20654	80,3 %
Dos	435	365					45,63 %
U2R							
Probe	7					4	0 %
R2L	11						0 %
Mandag uke 7							
	Normal	**Dos**	**U2R**	**Probe**	**R2L**	**Ukjent**	**Resultat**
Normal	220143	68		119		661	99,6 %
Dos	7	1				1	11,1 %
U2R							
Probe	9			4		5	22,2 %
R2L	11						0 %

Som vi ser av Tabell 36 vil en del av pakkene som tidligere ble klassifisert som normaltrafikk nå havne i ukjent klassen. Det vil imidlertid være en avveining hvor mange pakker vi kan la havne i klassen ukjent, samtidig som det helst ikke skal forekomme at angrepstrafikk blir klassifisert som normaltrafikk. Denne balansegangen kan til en viss grad styres av hvor stort avvik fra gjennomsnittsverdien til en klasse vi tolererer at en pakke kan ha før den blir klassifisert som ukjent.



Figur 12. Justering av avvik fra normalverdi for en klasse.

Figur 12 viser hvordan vi kan optimalisere forholdet mellom antall feilklassifiseringer og antall pakker som blir klassifisert som ukjent. Kurven ”Feilklassifiseringer” er en fellesbetegnelse for summen av pakker som blir klassifisert feil og falske alarmer som bør være lavets mulig. Kurven ”Ukjent” er antall pakker som blir klassifisert som ukjent. Disse pakkene oppfattes som falske alarmer og bør dermed også være lavest mulig. Av de to diagrammene vist i Figur 12 ser vi at det vi må godta over en faktor på 3 ganger gjennomsnittsverdi for en klasse for å få best mulig forhold mellom antall feilklassifiserte og antall pakker klassifisert som ukjent.

Ser vi på resultatene på samme måte som IDS'ene ble evaluert i DARPA Off-Line Intrusion Detection Evaluation, der man detekterer instanser av angrep og ikke nødvendigvis alle pakkene i hvert angrep, blir bildet annerledes. Hvis man går igjennom antall instanser av angrep og ser om klassifikatoren har detektert 1 eller flere pakker i instansen vil det se slik ut.

Tabell 37. Deteksjon av angrepsinstanser

	Antall instanser av angrep	Detektert	Resultat
Dos	5	4	80%
Probe	6	5	83,3 %
U2R	3	0	0%
R2L	2	0	0%

Tabell 37 viser hvor mange angrepsinstansene som detekteres når det klassifiseres ved å si at alt som ikke er angrep er normaltrafikk. I disse tilfellene vil angrepene som ikke detekteres slippe igjennom og utgjøre en fare for maskinen.

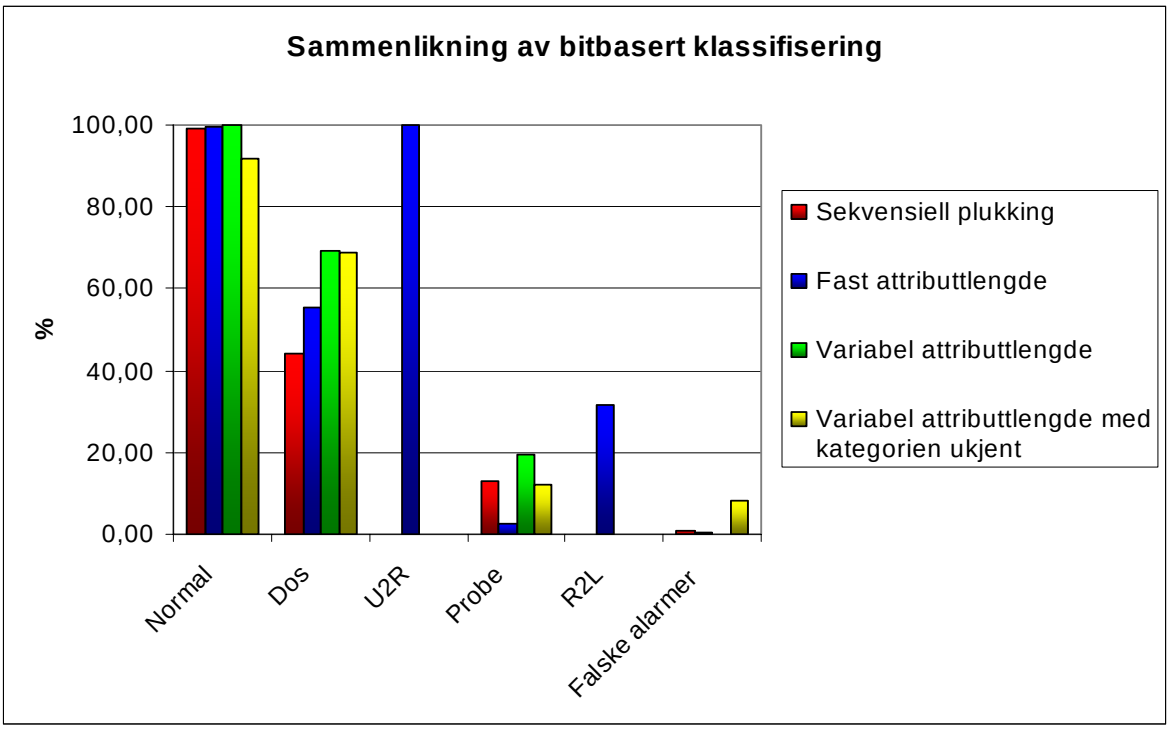
Tabell 38. Deteksjon av angrepsinstanser med trafikklassen ukjent

	Antall instanser av angrep	Detektert	Ukjent	Resultat
Dos	5	4	1	100%
Probe	6	3	3	100 %
U2R	3	0	2	67%
R2L	2	0		0%

Tabell 38 viser at innføringen av trafikklassen ukjent fører til at noen flere angrep kan stoppes. Det vil skje på den måten at pakker som havner i klassen ukjent ikke vil slippe inn til maskinen slik det gjør hvis pakkene havner i normal klassen. Så hvis vi går ut ifra resultatet i Tabell 38 ser vi at det er en høy grad av deteksjon av angrepsinstanser.

8 Drøfting

Legger vi resultatene som er beskrevet i kapittel 7 til grunn ser vi fort at det er noen måter å gruppere attributter på som gir bedre resultater enn andre. Det viser seg at det å bruke feltene i IP headeren sammen med TCP/UDP-headeren gir meget dårlig deteksjon av angrepstrafikk. Den sannsynlige grunnen er til at disse måtene ikke er gode nok, er størrelsen på attributtene. Særlig de store 32 bits attributtene er et problem da det kreves en enorm mengde treningsdata for å dekke alle muligheter som et slikt attributt i teorien kan anta. Filteret som innføres forbedrer resultatet noe i tillegg til at det balanserer forholdet mellom mengden av normaltrafikk og angrepstrafikk ved at mye av nettverkstrafikken som ikke har noe å si for deteksjon av angrep filtreres.



Figur 13. Sammenliking av de bitbaserte klassifiseringene

Går vi derimot over til å dele inn pakken i attributter på 8 bits blir resultatene betraktelig bedre. Ved å plukke bitene sekvensielt fra pakken oppnår vi en langt bedre deteksjon av alle typer trafikk, og at deteksjonen av normaltrafikk ligger på over 99.5 %. Vi ser av Figur 13 at det allikevel er den dårligste i alle kategorier. De beste resultatene oppnåes ved å gruppere attributtene med variabel lengde, og la de inneholde bits som er avhengige av hverandre. Som vi ser av Figur 13 greier vi å oppdage 99,89 % av all normaltrafikk med denne metoden. Det betyr igjen at vi får 0,11 % falske alarmer. Dette tallet høres lite ut men når man klassifiserer 1 million pakker betyr det at det oppstår 1100 falske alarmer som må inspiseres manuelt. Vi ser også at metoden scorer dårlig på deteksjon av angrepspakker. Hvor kritisk dette er for et

innbruddsdeteksjonssystem avhenger av hvilke klasse de feilklassifiserte pakkene plasseres. Hvis de plasseres i en annen angrepsklasse vil de allikevel bli sperret ute, mens det kritiske er hvis de plasseres i klassen normaltrafikk. Resultatene viser at det i stor grad er slik at de feilklassifiserte pakkene blir klassifisert nettopp som normaltrafikk. Dette er et resultat av at pakken er nødt å bli klassifisert til å tilhøre en av klassene. Dette problemet kan til en viss grad elimineres ved ta i bruk en klasse for alle pakker som ikke ser ut til å høre hjemme i noen av de andre klassene. Hva som skal til for ikke å høre i en klasse er et definisjonsspørmål. Vi definerer at sannsynligheten for at det er en bestemt klasse skal være innenfor en viss grense definert ut ifra gjennomsnittet av sannsynligheter for pakker som allerede er bestemt for klassen. Ved å justere denne grensen opp vil vi tillate større avvik fra gjennomsnittsverdien for klassen. Dette fører til at antall kritiske feilklassifiseringer øker, mens antall falske alarmer synker. Dette betyr at vi må prøve å finne balansen mellom kritiske feilklassifiseringer og falske alarmer. I Figur 12 prøver å vise hvordan vi kan vurdere hvor man skal sette grensen for å få justert klassifikatoren til best mulig ytelse.

Skal vi sammenlikne resultatene som er oppnådd i denne oppgaven med andre tilsvarende forsøk på Darpa datasettet må vi telletreff på en litt annen måte. Det heter seg at for å oppdage et angrep i Darpa datasettet holder det å detektere en av pakkene i angrepet. Vi har telt opp hvor mange av angrepsinstansene som faktisk detekteres i Tabell 37 og Tabell 38.
Ser vi på resultatene i Tabell 38 og sammenlikner dem med resultatene som ble oppnådd i DARPA Off-Line Intrusion Detection Evaluation[2], ligger treffprosenten rimelig høyt. I DARPA Off-Line Intrusion Detection Evaluation greide det beste systemet å detektere 55 % av angrepene ved å godta 100 falske alarmer. Ser vi på NETAD greide den å detektere 71 % av angrepene ved 100 falske alarmer. Øker vi toleransen for falske alarmer går også treffprosenten opp, og ved 5000 falske alarmer greier NETAD i overkant av 80 %. Dette er ikke direkte sammenlignbart med resultatene som blir presentert i denne oppgaven, men det gir en god indikasjon på at de beste resultatene i denne oppgaven er på høyde med hva NETAD har oppnådd.

Det er også rimelig og tro at vi ved å øke mengden treningsdata vil være i stand til å få en enda bedre treffprosent. Dette antar vi fordi vi ser det oppnåes bortimot 100 % korrekt deteksjon på trafikk som er et utvalg av treningsdataene. Ut ifra dette kan det se ut som det er noe mangel på treningsdata i enkelte trafikklasser som gjør at resultatene faller noe på klassifisering av data som ikke er en del av treningssettet. Dette blir ganske tydelig da det er på normaltrafikk vi ser de beste resultatene, samtidig som det er av klassen normal det eksister suverent mest treningsdata.
Det kan også knyttes en viss usikkerhet til om merkingen av angrepstrafikk er korrekt, noe som i tilfelle virker inn under trening av klassifikatoren. Merkingen er utført manuelt ved å bruke attributter som er unike for angrepssesjonen, noe som kan føre til at enkelte pakker blir feilmerket. I store mengder treningsdata vil dette bli eliminert som støy, men i små mengder treningsdata, som vi har i enkelte av trafikklassene, kan det ha relativt store utslag i form at vi kan få feilklassifiseringer.

9 Konklusjon

Vi har i denne rapporten vist at Naiv Bayesiansk klassifikator kan brukes for å lage en effektiv klassifikator for nettverkstrafikk. Det har blitt testet 4 metoder for utvelgelse av attributter for bruk i klassifikatoren og vi har sett at det har stor betydning for nøyaktigheten til klassifikatoren hvordan disse attributtene velges. Ved å kombinere filtrering av nettverkstrafikk samt å bruke attributter med variabel lengde er det mulig å oppnå resultater som er på fullt høyde med andre sammenliknbare arbeider. Et viktig poeng med de variable attributtene er at de inneholder bits som er avhengige av hverandre. Med å innføre denne avhengigheten beholder vi spesielle mønstre blant bits i en trafikklasse som er karakteristiske for denne klassen. Dette bidrar til bedre resultater.
Antall falske alarmer som genereres kan holdes på et kontrollerbart nivå. Antall falske alarmer øker med høyere grad av deteksjon av angrep, men vi viser hvordan forholdet mellom falske alarmer og feilklassifiseringer kan justeres for å finne det optimale forholdet.

Resultatene vi oppnår viser at vi med en enkel læringsalgoritme er i stand til å oppnå gode resultater. Dette er lovende da den Naive Bayesianske klassifikatoren er effektiv med hensyn til hvordan den trenes. Mer treningsdata vil gi oss bedre resultater, uten at det går ut over kompleksiteten til klassifikatoren. Det betyr at vi kan forbedre resultatene i ettertid ved å tilføre klassifikatoren mer treningsdata.

Et av problemene til klassifikatoren slik den fremstår nå er at den ikke vil bli bedre etter hvert som den ukjente datatrafikken behandles. Ved å behandle denne trafikken manuelt og la klassifikatoren få muligheten til å trene seg opp igjen på bakgrunn av denne manuelle klassifiseringen vil klassifikatoren kunne få stadig bedre ytelse. Dette vil hjelpe på lik linje ved å tilføre klassifikatoren mer treningsdata, og vil være neste naturlige utvidelse for å lage en god klassifikator for nettverkstrafikk.
I tillegg ville det være interessant å kjøre en full opptrening og test mot Darpa datasettet slik at resultatene var direkte sammenliknbare med hva for eksempel NETAD har oppnådd. Dette krever litt omstrukturering av hvordan systemet trenes slik at prosessen ikke blir like tidkrevende og manuell slik den er nå, men resultatene ville være minst på høyde med hva NETAD presterer.

10 Referanser

[1] Mahoney, M: Network Traffic Anomaly Detection Based on Packet Bytes

[2] Richard Lippmann, Joshua W. Haines, David J. Fried, Jonathan Korba, Kumar Das, The 1999 DARPA Off-Line Intrusion Detection Evaluation Lincoln Laboratory MIT, 244 Wood Street, Lexington, MA 02173-9108

[3] Lincoln Laboratory, MIT, http://www.ll.mit.edu/IST/ideval/data/data_index.html

[4] Meila, Marina: An Accelerated Chow and Liu Algorithm: Fitting Tree Distributions to High-Dimensional Sparse Data, MIT Cambridge, MA 02142, USA

[5] Paxson, Vern, "Bro: A System for Detecting Network Intruders in Real-Time", Lawrence Berkeley National Laboratory Proceedings, 7'th USENIX Security Symposium,Jan. 26-29, 1998, San Antonio TX,

[6] Roesch, Martin, "Snort - Lightweight Intrusion Detection for Networks", Proc. USENIX Lisa '99, Seattle: Nov. 7-12, 1999.

[7] SPADE, Silicon Defense, http://www.silicondefense.com/software/spice/

[8] Sekar, R., M. Bendre, D. Dhurjati, P. Bollineni, "A Fast Automaton-based Method for Detecting Anomalous Program Behaviors". Proceedings of the 2001 IEEE Symposium on Security and Privacy

[9] Anderson, D. et. al., "Detecting unusual program behavior using the statistical component of the Next-generation Intrusion Detection Expert System (NIDES)", Computer Science Laboratory SRI-CSL 95-06 May 1995. http://www.sdl.sri.com/papers/5/s/5sri/5sri.pdf

[10] Mahoney, M., P. K. Chan, "PHAD: Packet Header Anomaly Detection for Identifying Hostile Network Traffic", Florida Tech. technical report 2001-04, http://cs.fit.edu/~tr/

[11] Mahoney, M., P. K. Chan, "Learning Models of Network Traffic for Detecting Novel Attacks", Florida Tech. technical report 2002-08, http://cs.fit.edu/~tr/

[12] Mahoney, M., P. K. Chan, "Learning Nonstationary Models of Normal Network Traffic for Detecting Novel Attacks ", Edmonton, Alberta: Proc. SIGKDD, 2002, 376-385.

[13] RFC 791: Internet Protocol, ftp://ftp.rfc-editor.org/in-notes/rfc791.txt

[14] RFC 792: Internet Control Message Protocol, ftp://ftp.rfc-editor.org/in-notes/rfc792.txt

[15] RFC 768: User Datagram Protocol, ftp://ftp.rfc-editor.org/in-notes/rfc768.txt

[16] RFC 793: Transmission Contol Protocol, ftp://ftp.rfc-editor.org/in-notes/rfc793.txt

11 Vedlegg

Vedlegg A - Kildekode for bit-avhegighets algoritme.

11.1 Vedlegg A – Kildekode for bit-avhegighets algoritme

```
import math
import cPickle
class Dependency:
	# Table for counting the number of times the bits of each attribute pair are set in the same entry
	nuv = None
	#Array for counting the number of times the bit of an attribute is set
	n = None
	# Tuple for storing the bits of a given entry
	e = None
	#the input
	input = None
	#Table for storing the mutual information of each attribute pair
	iuv = None
	candidate_edge = None
	candidate_edge_dict = {}
	# Number of attributes
	ATTRIBUTES = 384
	# Number of entries
	ENTRIES = 16
	# Number of bits in each group
	BITSINGROUP = 8
	#stores attributes in a group
	attributesgroup = []
	# stores the finished groups
	grouped_attributes = {}
	#attributes prosessed
	excludelist = []
	#stores attributs prosessed
	attributes_prosessed = 0
	#treshold for grouping bytes
	treshold = 0.01
	ttypedict={}

	def __init__(self):
		self.nuv = {}
		self.n = {}
		self.iuv = {}
		self.candidate_edge = {}

	# Count number of set bits for each attribute as well as for each attribute pair
	def count_setbits(self, input, iteration, traffictype):
		if self.ttypedict.has_key(traffictype):
			self.ENTRIES = self.ttypedict[traffictype][0]
			self.n = self.ttypedict[traffictype][1]
			self.nuv = self.ttypedict[traffictype][2]
		#sets some initial values for counting
		self.input = input
		ent = len(self.input)
		self.ENTRIES += len(self.input)
		self.ATTRIBUTES = len(self.input[0])
		#Initialize bit- and bit pair counters
		if iteration == 1:
			u = 0
			while u < self.ATTRIBUTES:
				self.n[u] = 0
				v = u+1
				while v < self.ATTRIBUTES:
					self.nuv[(u,v)] = 0
					self.iuv[(u,v)] = 0
					v += 1
				u+=1
		#Go through each entry and update counters
		i = 0
		while i < ent: #self.ENTRIES:
			self.e = self.input[i]
			# Update bit counters and bit pair counters
			u = 0
			while u < self.ATTRIBUTES:
				if self.e[u] == 1:
					self.n[u] += 1
					# Counts number of bit pair combinations
```

```
                        v = u+1
                        while v < self.ATTRIBUTES:
                            if self.e[v] == 1:
                                self.nuv[(u,v)] += 1
                            v += 1
                    u+=1
                i+=1
            self.ttypedict[traffictype] = (self.ENTRIES,self.n,self.nuv)
        #Calculate mutual information with respect to each bit-pair
        def calculate_mutual_information(self):
            for traffictype,value in self.ttypedict.iteritems():
                self.ENTRIES = value[0]
                self.n = value[1]
                self.nuv = value[2]
                self.candidate_edge = {}
                pos = 0
                u= 0
                v = None
                su = 0
                sv = 0
                puv = {(0,0):0,(0,1):0,(1,0):0,(1,1):0}
                pu = {0:0,1:0}
                pv = {0:0,1:0}
                while u < self.ATTRIBUTES:
                    v = u+1
                    while v < self.ATTRIBUTES:
                        # Calculate the relevant probabilities
                        puv[(1,1)] = float(self.nuv[(u,v)]) / float(self.ENTRIES)
                        puv[(1,0)] = (float(self.n[u]) - float(self.nuv[(u,v)])) / float(self.ENTRIES)
                        puv[(0,1)] = (float(self.n[v]) - float(self.nuv[(u,v)])) / float(self.ENTRIES)
                        puv[(0,0)] = (float(self.ENTRIES) - float(self.n[u]) - float(self.n[v]) +
float(self.nuv[(u,v)])) / float(self.ENTRIES)
                        pu[1] = float(self.n[u]) / float(self.ENTRIES)
                        pu[0] = float(self.ENTRIES - self.n[u]) / float(self.ENTRIES)
                        pv[1] = float(self.n[v]) / float(self.ENTRIES)
                        pu[0] = float(self.ENTRIES - self.n[v]) / float(self.ENTRIES)
                        # Calculate mutual information
                        while su <= 1:
                            while sv <= 1:
                                if puv[(su,sv)] > 0 and pu[su] > 0 and pv[sv] > 0 :
                                    self.iuv[(u,v)] += float(puv[(su,sv)]) * float(math.log(
float(puv[(su,sv)]) / float(pu[su] * pv[sv]) ))
                                sv += 1
                            sv = 0
                            su += 1
                        su = 0
                        self.iuv[(v,u)] = self.iuv[(u,v)]
                        # Store produced information about edges in edge array
                        self.candidate_edge[pos] = (u,v,self.iuv[(u,v)])
                        pos += 1
                        v +=1
                    u+=1
                self.candidate_edge_dict[traffictype] = self.candidate_edge
            #just storing the calculated information
            try:
                file = open ("calculated_dict.txt",'w')
                cPickle.dump(self.candidate_edge_dict,file)
                file.close()
            except:
                file = open ("d:\\calculated_dict.txt",'w')
                cPickle.dump(self.candidate_edge_dict,file)
                file.close()
        #makeing the attribute groups
        def make_dependent_groups(self, fixed_groups = True):
                #use the stored claculated infoamtion
                file = open("calculated_dict.txt",'r')
                self.candidate_edge_dict = cPickle.load(file)
                file.close()
                # This is used for fixed 8 bit attributes
                if fixed_groups == True:
                    #initializes som variables
                    self.candidate_edge = self.candidate_edge_dict[1]
                    number_of_groups = 0
                    groupcounter = 0
                    bitcounter = 0
                    max = -1000000000000000000000000
```

```
                        pos = -1
                        attributes_processed = []
                        attributegroup = []
                        # calculates the number of attrbutes needed
                        if self.ATTRIBUTES % self.BITSINGROUP == 0:
                            number_of_groups = self.ATTRIBUTES / self.BITSINGROUP
                        else:
                            number_of_groups = (self.ATTRIBUTES / self.BITSINGROUP) + 1
                        #until all groups are full;)
                        while groupcounter < number_of_groups:
                            max = -100000000000000000000000
                            pos = -1
                            #counter for number of bits in attribute
                            bitcounter = 0
                            #finds the maximum dependency
                            for posinfo, attribute in self.candidate_edge.iteritems():
                                    if attribute[2] >= max and (attribute[0] not in attributes_processed
and attribute[1] not in attributes_processed ):
                                        max = attribute[2]
                                        pos = posinfo
                            #stores the bitpair with highest dependency
                            a = self.candidate_edge[pos][0]
                            b = self.candidate_edge[pos][1]
                            attributegroup.append(a)
                            bitcounter += 1
                            attributegroup.append(b)
                            bitcounter += 1
                            self.deletepos(pos)
                            #adds bit with the highest dependency with respect to the 2 already existing
in group
                            while bitcounter < self.BITSINGROUP:
                                max = -100000000000000000000000
                                pos = -1
                                #find pos of highest related
                                for posinfo, attribute in self.candidate_edge.iteritems():
                                    if attribute[2] >= max and (attribute[0] in attributegroup or
attribute[1] in attributegroup) and (attribute[0] not in attributes_processed and attribute[1] not in
attributes_processed ):
                                        max = attribute[2]
                                        pos = posinfo
                                a = self.candidate_edge[pos][0]
                                b = self.candidate_edge[pos][1]
                                if a not in attributegroup:
                                    attributegroup.append(a)
                                    bitcounter += 1
                                if b not in attributegroup:
                                    attributegroup.append(b)
                                    bitcounter += 1
                                self.deletepos(pos)
                            self.grouped_attributes[groupcounter] = attributegroup
                            attributes_processed += attributegroup
                            attributegroup = []
                            groupcounter += 1

                #this is used for grouping by the dependency beetween bits - not nescessary ith 8 bit
in each attribute
                if not fixed_groups:
                    #initial max dependency
                    max = -100000000000000000000000
                    #position of max dependency
                    pos = -1
                    #which attributes are already processed
                    attributes_processed = []
                    #the grouped attributes
                    attributegroup = []
                    bitcounter = 0
                    groupcounter = 0
                    #as long as not all 384 bits are processed
                    while len(attributes_processed) < 384:
                        # if all but 1-bit are processed
                        if len(attributes_processed) == 383:
                            i = 0
                            while i<384:
                                if i not in attributes_processed:
                                    attributegroup.append(i)
                                    break
```

```
                        i+=1
                else:
                    max = -1000000000000000000000
                    pos = -1
                    tt = 0
                    bitcounter = 0
                    #finds the maximum dependency
                    for key,dict in self.candidate_edge_dict.iteritems():
                        for posinfo, attribute in dict.iteritems():
                            if float(attribute[2]) >= float(max) and (attribute[0] not in
attributes_processed and attribute[1] not in attributes_processed ):
                                max = float(attribute[2])
                                pos = posinfo
                                tt = key
                    #if the dependency is greater than the predefined threshold of
dependency
                    if float(max) > float(self.treshold):
                        a = self.candidate_edge_dict[tt][pos][0]
                        b = self.candidate_edge_dict[tt][pos][1]
                        #adds the edges to the attributegoup
                        attributegroup.append(a)
                        bitcounter += 1
                        attributegroup.append(b)
                        bitcounter += 1
                        #deletes the attribute
                        self.deletepos(pos,tt)
                    else: #the two bits are not dependent, (dependency below the
threshold) - they er splitted
                        a = self.candidate_edge_dict[tt][pos][0]
                        attributegroup.append(a)
                        bitcounter += 1
                        #deletes the attribute
                        self.deletepos(pos,tt)

                    #fills up the group as long as there ar bit related to one in the group
with denpendencey above the threshold, (maximum 8 bit in group)
                    while bitcounter < self.BITSINGROUP:
                        max1 = -1000000000000000000000
                        pos1 = -1
                        tt = 1
                        #find pos of highest related
                        for traffictype,dict in self.candidate_edge_dict.iteritems():
                            for posinfo, attribute in dict.iteritems():
                                if float(attribute[2]) >= float(max1) and (attribute[0] in
attributegroup or attribute[1] in attributegroup) and (attribute[0] not in attributes_processed and
attribute[1] not in attributes_processed ):
                                    max1 = float(attribute[2])
                                    pos1 = posinfo
                                    tt = traffictype

                        #if the highest related max value is less than the predefined
treshold - dont put more bits in the group
                        if float(max1) > float(self.treshold):
                            #adds the found edges to the attributegroup if they not
already exist in group
                            a = self.candidate_edge_dict[tt][pos1][0]
                            b = self.candidate_edge_dict[tt][pos1][1]
                            if a not in attributegroup:
                                attributegroup.append(a)
                                bitcounter += 1
                            if b not in attributegroup:
                                attributegroup.append(b)
                                bitcounter += 1
                            #delete the processed attribute
                            self.deletepos(pos1,tt)
                        else:#no more bits with dependency above threshold
                            break
                #saves the attribute infomation
                self.grouped_attributes[groupcounter] = attributegroup
                attributes_processed += attributegroup
```

```
                        attributegroup = []
                        groupcounter += 1

    def deletepos(self,pos,tt):
        self.candidate_edge_dict[tt].pop(pos)



    def getHighestProbRelated(self,ag):
        #for each attribute already found
        max = 0
        pos = -1

        for item in ag:
            # the greatest probability with respect to items in the list
            for posinfo, attribute in self.candidate_edge.iteritems():
                #if the pos is related to the item

                if attribute[0] == item or attribute[1] == item:

                    val = self.candidate_edge[posinfo][2]

                    if val >= max:

                        max = val
                        pos = posinfo
        print "pos to be added: "+str(pos)
        max = 0
        #print self.candidate_edge
        #try:
        if 1:
            a = self.candidate_edge[pos][0]
            b = self.candidate_edge[pos][1]

            if a not in ag:
                ag.append(a)
                self.excludelist.append(a)
            if b not in ag:
                ag.append(b)
                self.excludelist.append(b)
            self.deletepos(pos)
        #except KeyError:
            #pass
        return ag

if __name__ == "__main__":
    inp =
((1,1,0,1,1,1,1,1,1,1,0,1,1,0,1,1,1,1,0,1,1,1,1,1,1,1,0,1,1,0,1,1),(1,0,1,1,1,1,1,0,1,1,0,1,0,1,1,0,1,1,0,1,1,0,1,1
    #inp = ((1,1,0,1),(1,1,0,0),(1,1,0,0),(1,1,0,0))
    d = Dependency(inp)
    d.count_setbits()
    #print "n: "+str(d.n)
    #print "nuv: "+str(d.nuv)

    d.calculate_mutual_information()

    #print "candidate_edge: "#+str(d.candidate_edge)
    #for m,n in d.candidate_edge.iteritems():
    #    print str(m)+":"+str(n)
    d.make_dependent_groups()
    #print d.grouped_attributes
    """
    /* Table for counting the number of times the bits of each attribute
pair are set in the same entry */
int nuv[ATTRIBUTES][ATTRIBUTES];
/* Array for counting the number of times the bit of an attribute is
set */
int n[ATTRIBUTES];
/* Array for storing the bits of a given entry */
int e[ATTRIBUTES];
/* Table for storing the mutual information of each attribute pair */
float iuv[ATTRIBUTES][ATTRIBUTES];
/* Count number of set bits for each attribute as well as for each
attribute pair */
void count_set_bits()
{ int u, v, i;
```

```
/* Initialize bit- and bit pair counters */
for (u = 0; u < ATTRIBUTES; u++) {
n[u] = 0;
for (v = u + 1; v < ATTRIBUTES; v++) {
	nuv[u][v] = 0;
}
}
/* Go through each entry and update counters */
for (i = 0; i < ENTRIES; i++) {
	generate_entry();
	/* Update bit counters and bit pair counters */
	for (u = 0; u < ATTRIBUTES; u++) {
		if (e[u] == 1) {
			n[u] += 1;
			/* Counts number of bit pair combinations */
			for (v = u + 1; v < ATTRIBUTES; v++) {
				if (e[v] == 1)
				nuv[u][v] += 1;
			}
		}
	}
}
}
/***** Calculate mutual information with respect to each bit-pair *****/
void calculate_mutual_information()
{
int pos, u, v, su, sv;
float puv[2][2], pu[2], pv[2];
pos = 0;
for (u = 0; u < ATTRIBUTES; u++) {
for (v = u + 1; v < ATTRIBUTES; v++) {
	/* Calculate the relevant probabilities */
	puv[1][1] = 1.0*nuv[u][v]/ENTRIES;
	puv[1][0] = 1.0*(n[u] - nuv[u][v])/ENTRIES;
	puv[0][1] = 1.0*(n[v] - nuv[u][v])/ENTRIES;
	puv[0][0] = 1.0*(ENTRIES - n[u] - n[v] + nuv[u][v])/ENTRIES;
	pu[1] = 1.0*(n[u])/ENTRIES;
	pu[0] = 1.0*(ENTRIES - n[u])/ENTRIES;
	pv[1] = 1.0*(n[v])/ENTRIES;
	pv[0] = 1.0*(ENTRIES - n[v])/ENTRIES;
	/* Calculate mutual information */
	iuv[u][v] = 0.0;
	for (su = 0; su <= 1; su++) {
	for (sv = 0; sv <= 1; sv++) {
		if (puv[su][sv] > 0 && pu[su] > 0 && pv[sv] > 0) {
		iuv[u][v] += puv[su][sv]*log(puv[su][sv]/(pu[su]*pv[sv]));
		} }
	}
	iuv[v][u] = iuv[u][v];
	/* Store produced information about edges in edge array */
	candidate_edge[pos].a = u;
	candidate_edge[pos].b = v;
	candidate_edge[pos].i = iuv[u][v];
	pos++;
}
}
}
```