Datavetenskap

Mikael Hackman **Johan Strandbergh**

Partiell nätverkssäkerhet med inriktning på multimedia

Magisteruppsats

2003:04

Partiell nätverkssäkerhet med inriktning på multimedia

Mikael Hackman **Johan Strandbergh**

Denna uppsats är skriven som en del av det arbete som krävs för att erhålla en magisterexamen i datavetenskap. Allt material i denna rapport, vilket inte är mitt eget, har blivit tydligt identifierat och inget material är inkluderat som tidigare använts för erhållande av annan examen.

Mikael Hackman

Johan Strandbergh

Godkänd, 2003-06-17

Handledare: Thijs Holleboom

Examinator: Donald F. Ross

Sammanfattning

Det finns flera protokoll såsom SSL och IPSec som underlättar hanteringen för användaren av de säkerhetskrav som kan uppstå i nätverksmiljöer. De saknar tyvärr mycket av den dynamik som skulle önskas för att kunna erbjuda en lättanvänd konfigurerbar säkerhetslösning som lätt kan anpassas till användarens krav på både säkerhet och prestanda.

För att kunna öka prestandan hos en given krypteringsalgoritm måste antingen algoritmen i sig förbättras, alternativt dess användande. Eftersom det redan pågår kontinuerlig forskning på att förbättra algoritmerna och detta har gjorts under flera år är det användandet av algoritmen som är det intressanta att titta på. Av denna anledningen har vi arbetat fram ett koncept som vi kallar för lättviktskryptering, vilket tillsammans med starka krypteringsalgoritmer leder till konceptet partiell säkerhet. Efter att idén presenterats visas resultatet av prestandatester baserade på konventionella krypteringsmetoder och partiell kryptering. För speciella filformat, som till exempel JPEG, introduceras begreppet riktad kryptering med idéer till dess användningsområde och fördelar.

Partial network security directed towards multimedia

SSL and IPSec are two examples of existing protocols that make it easier for the user to handle some of the security demands that can appear in a networking environment. Unfortunately they lack much of the flexibility that is necessary to offer an easy to use, dynamic security solution that can be configured to meet both the security demands from the user, as well as the performance demands.

In order to increase the performance for a given encryption algorithm, either the algorithm itself can be improved, or the way it is used. Research on improving the algorithms themselves has been going on for several years. In this work we focus instead on the usage of algorithms. Due to this we propose a concept we call lightweight encryption as an solution, which together with conventional encryption algorithms has brought forward the concept of partial security. When this idea has been introduced, tests are presented that show the difference in performance between conventional encryption and partial encryption. For special file formats, as for example JPEG, we introduce the concept of directed encryption along with further ideas of usage and advantages of partial security.

Tack

Vi önskar tacka följande personer för deras hjälp med denna uppsats:

- Stefan Lindskog och Thijs Holleboom, våra handledare. För all tid de lagt ner på oss och möjligheten att skriva denna uppsats. För att de stod ut med vårt gnällande och såg till att vi höll oss på rätt spår.
- Hans Hedbom, för hans idéer och framför allt lugnande ord när allt verkade hopplöst.
- Johan Garcia, för hans hjälp med JPEG och samtal som inspirerade oss till den riktade krypteringen.
- Övriga personer på institutionen för informationsteknologi, samt Igor Gachkov, som stått ut med våra frågor i tid och otid.

Innehåll

Figurer

Tabeller

Kapitel 1

Introduktion

Internet bygger på principen "best effort". Det vill säga att inga garantier ges för den trafik som skickas via nätverket. Det har dock på senare tid dykt upp applikationer som ställer speciella krav på trafiken. För till exempel en videokonferens är det viktigt att data snabbt når mottagaren. I detta fall kanske vissa förlorade paket kan tolereras. För en filöverföring kanske användaren kan nöja sig med en längre fördröjning och minskad bandbredd, så länge som alla paket kommer fram. Quality of Service (QoS) är ett sätt att hantera de prestandakrav som finns. Detta ger möjlighet att reservera resurser i nätverket för att erbjuda den service som krävs av applikationen.

Trafiken sker oftast via komplexa nätverk och säkerheten är något som måste sättas i fokus för att kunna kontrollera att obehörigt inblandande inte sker. Kraven på säkerhet kan i många fall påverka prestandan på ett negativt sätt. Ibland måste därför en avvägning göras mellan prestanda och säkerhetskrav. Om användaren har krav på säker överföring så finns i dagens läge inga möjligheter att hantera dessa krav tillsammans med prestandakrav och liknande. Säkerhet ses i dagens läge för det mesta som något binärt. Antingen anses överföringen vara helt säker eller så saknas säkerheten helt. För att kunna förhandla fram önskad säkerhetsnivå och anpassa denna till övriga krav måste säkerheten kunna hanteras på ett mera dynamiskt sätt.

1.1 Avgränsningar

Vi kommer att undersöka vad som krävs för att kunna erbjuda mera dynamiska säkerhetslösningar i de metoder och mekanismer som används för att kunna erbjuda en säker kommunikation via nätverk.

Då den idé om partiell säkerhet som vi introducerar i kapitel 3.4 kan appliceras på verifieringen av meddelanden utan att behöva ta hänsyn till vilken typ av data det sker på kommer inte vidare tester av detta att utföras. Vi kommer i stället att koncentrera oss på hur dessa eller liknande tekniker kan appliceras på krypteringen av meddelanden.

Den metod vi presenterar för lättviktskryptering i kapitel 4.4 har tagits fram beroende på dess snabbhet. Då denna inte behöver erbjuda ett lika starkt skydd som en konventionell krypteringsalgoritm kommer vi enbart att visa att med den metod vi använder kan säkerheten varieras på ett dynamiskt sätt. Vi kommer dock att beskriva några av de kända svagheterna i metoden.

1.2 Mål

Vi ska undersöka några av de vanligaste metoder som idag används för att hantera säker kommunikation över nätverk. För att uppnå önskade möjligheter till konfigurerbar säkerhet uppfyller de undersökta metoderna inte alla de krav som vi ställer. Vi ska därför utveckla ett sätt att skapa en dynamisk säkerhetsnivå som kan implementeras i dessa applikationer.

Då de största behoven verkar vara att kunna anpassa kryptering på ett dynamiskt sätt har vi lagt tyngdpunkten inom detta område. Vårt mål är att undersöka möjligheten att uppnå prestandavinster genom att sänka kraven på säkerheten. Detta ska kunna ske på ett dynamiskt sätt för att kunna anpassas till flera tänkbara situationer.

1.3 Sammanfattning

Bakgrunden till arbetet beskrivs i kapitel 2. Vi undersöker där relaterade arbeten, samt ger en modell för hur säkerhet kan beskrivas. Vi ger även en kort introduktion till begreppet QoS.

Därefter beskrivs i kapitel 3.1 några av de metoder som i dagens läge används för att hantera säker kommunikation. Vi tar där upp de brister som gör att de inte kan integreras som en del av QoS i befintligt utförande. I kapitel 3.3 koncentrerar vi oss på de algoritmer som används för kryptering i dessa applikationer. Vi beskriver där kortfattat hur några av dessa fungerar och hur de påverkar prestandan för dataöverföringen.

Efter att ha undersökt befintliga säkerhetsprotokoll och krypteringsalgoritmer introducerar vi i kapitel 3.4 sedan en idé till hur säkerheten kan varieras genom mindre modifikationer i befintliga applikationer. Dessa modifikationer består av att endast hantera en delmängd av datan på ett säkert sätt.

För att på ett generellt sätt kunna hantera kryptering på detta sätt behövs en enkel och resurssnål krypteringsalgoritm. En sådan algoritm utvecklas och beskrivs i kapitel 4.4 och har visat sig vara nödvändig för att ge ett grundskydd för all data som skickas.

För filer där filstrukturen är känd är grundskyddet inte alltid nödvändigt och i kapitel 4.5 undersöker vi några av de möjligheter som erbjuds vid partiell kryptering när filens innehåll är känt.

För att kontrollera att krypteringsmetoden som tagits fram i kapitel 4.4 reducerar belastningen har prestandatester utförts. Kapitel 5 beskriver testutförandet och resultaten presenteras sedan i kapitel 6.

Hela arbetet sammanfattas slutligen i kapitel 7.

Kapitel 2

Bakgrund

Dagens applikationer blir allt mer prestandakrävande. Dessutom ökar också kraven på säkerhet. Om säkerhetskraven ska uppfyllas medför även detta en förändring i prestanda. Säkerhet och prestanda hanteras i dagens läge separerade från varandra. Vi ska i detta kapitel titta på hur prestandakrav hanteras i QoS. Därefter ska vi beskriva en generell modell för hur säkerheten hanteras för nätverkstrafik.

2.1 Projektets ursprung

Stefan Lindskog och Erland Jonsson vid Chalmers tekniska högskola har skrivit en artikel [LJ02] om hur säkerhet skulle kunna integreras som en dimension i QoS-arkitekturer. Liknande forskning bedrivs av Irvine och Levine [IL00] vid Naval Postgraduate School i Monterey, USA. De använder sig av benämningen Quality of Security Service (QoSS).

2.2 Quality of Service

Med QoS menas möjligheten för ett nätverk att tillhandahålla bättre och för ändamålet lämpligare service för utvald nätverkstrafik över flera typer av nät som Asynchronous

Transfer Mode (ATM), Ethernet, SONET och alla IP-baserade nätverk. Det primära målet med QoS är att erbjuda prioritering, inklusive reserverad bandbredd, kontrollerat jitter och latency. En annan viktig egenskap är att se till att prioriteringen av ett eller flera dataflöden inte får något annat flöde att haverera. En artikel som på ett bra sätt sammanfattar QoS har skrivits av Xiao och Ni [XN99].

Den lösning som QoS tillhandahåller för att erbjuda de tjänster som användaren begär innebär att förhandling sker för att reservera de resurser som krävs för önskad servicekvalitet. För närvarande finns inga möjligheter att specificera önskad säkerhet med befintliga serviceklasser. Om säkerhet introduceras som en dimension i QoS-arkitekturen, skulle användaren kunna förhandla om säkerhet på samma sätt som övriga parametrar. På samma sätt som användarens krav på ökad bild eller ljudkvalitet, kommer användarens krav på kryptering att leda till högre krav på systemet i form av processorkraft för att kunna leverera önskad kvalitet till användaren.

2.3 Säkerhet

Hur säkerheten i ett system hanteras beskrivs oftast av en säkerhetspolicy. En sådan policy beskriver vilka rättigheter en användare har samt hur systemet skyddas mot otillåtet användande. Så fort information skickas mellan användare över ett publikt nätverk som till exempel Internet kan inte systemet längre kontrolleras av användaren. Detta leder till att åtgärder för att skydda datan måste tas om denna ska kunna skyddas från obehörigt utnyttjande.

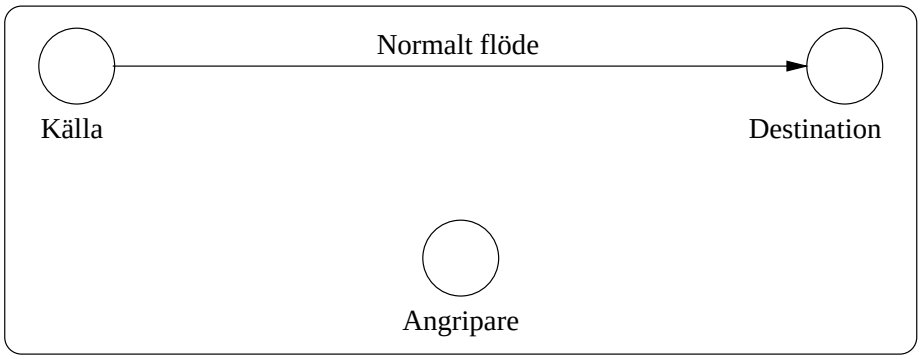
Vi ska i detta kapitel titta på vilka typer av attacker som kan utföras när data skickas mellan användare över ett publikt nätverk, samt vilka motåtgärder som kan utföras för att skydda den data som skickas. Säkerheten kan på så sätt beskrivas som olika attribut, där varje attribut beskriver vilken typ av säkerhet som uppnås från de åtgärder som utförs. Utifrån dessa attribut beskrivs sedan en modell för hur säker kommunikation kan uppnås

när data skickas över ett publikt nätverk.

2.3.1 Säkerhetsattribut

Ett angrepp kan ske på en mängd olika sätt beroende på vad en angripare är ute efter. Helt säker kommunikation kan endast ske när det inte finns några möjligheter för en angripare att över huvud taget upptäcka att kommunikationen sker. Om kommunikationen sker via ett publikt nätverk är detta helt omöjligt att uppnå. Vi måste därför beskriva på vilket sätt vi skyddar oss mot ett angrepp.

Vi ska i detta kapitel beskriva vilka attribut som kan användas för att beskriva hur säker kommunikation uppnås via ett osäkert nätverk. Önskvärt är att kommunikationen sker mellan sändare och mottagare utan någon som helst inblandning från en angripare. Normal kommunikation mellan sändare och mottagare illustreras i figur 2.1

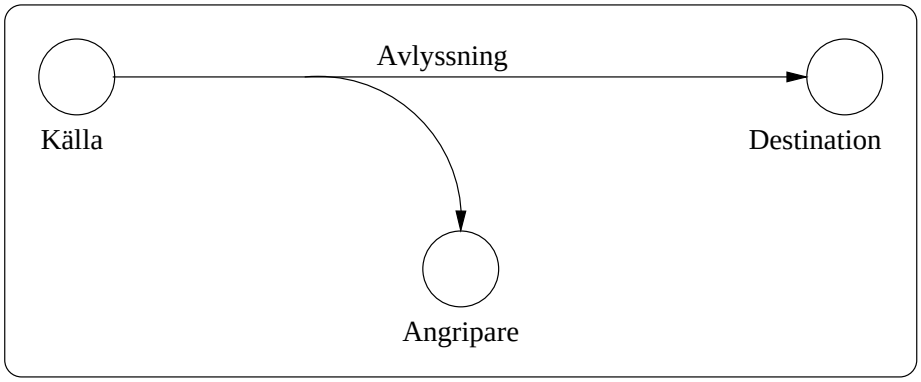


Figur 2.1: Normal kommunikation mellan två parter

Sekretess

Vid diskussioner om säkerhet är det ofta sekretess som det talas om. Sekretess betyder att den data som skickas hålls hemlig från alla utom sändaren och den avsedda mottagaren (eller mottagarna). När det gäller sekretess i datasammanhang är användandet av olika kryptografiska algoritmer det vanligaste sättet att skydda datan när den skickas mellan

två parter. Om datan krypteras så uppnås sekretess, vilket förhindrar avlyssning av den data som skickas. En attack mot sekretess illustreras i figur 2.2.

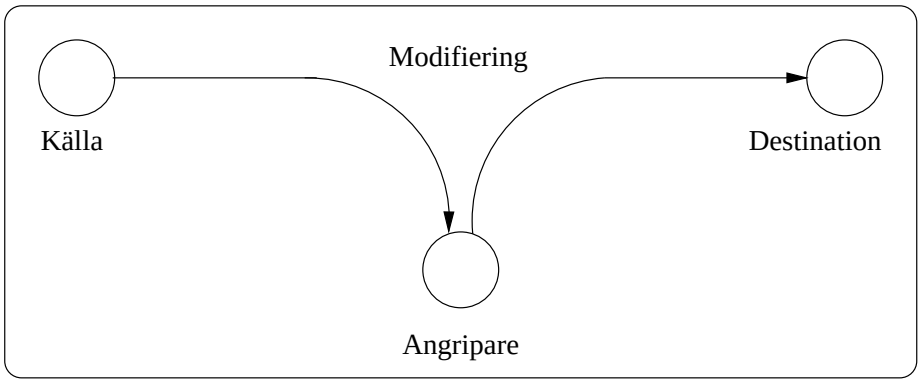


Figur 2.2: Avlyssnad kommunikation mellan två parter

Även om ett meddelande krypteras kan inte full sekretess uppnås. Trafikanalys kan fortfarande ske där en angripare erhåller information om var meddelanden skickas och med vilken intensitet trafiken sker. Det finns heller inget krypto som i dagsläget är garanterat oknäckbart, utan en viss risk finns alltid att en angripare kan få tillgång till datan från ett krypterat meddelande.

Integritet

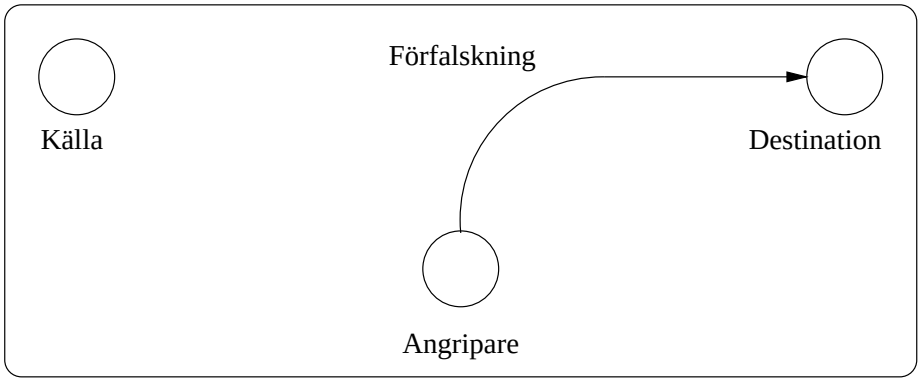
En mottagare vill kunna vara säker på att det är originalmeddelandet som skickades utan några ändringar på vägen. Med integritet menas att en mottagare direkt ska kunna se om meddelandet snappats upp och förändrats. Checksumman av meddelandet beräknas när det skickas. En checksumma måste antingen skyddas med kryptering så den inte kan ändras, alternativt så läggs den upp publikt så att mottagaren kan gå in på avsändarens hemsida och där läsa checksumman. En attack mot integriteten illustreras i figur 2.3.



Figur 2.3: Modifierad kommunikation mellan två parter

Autentisering

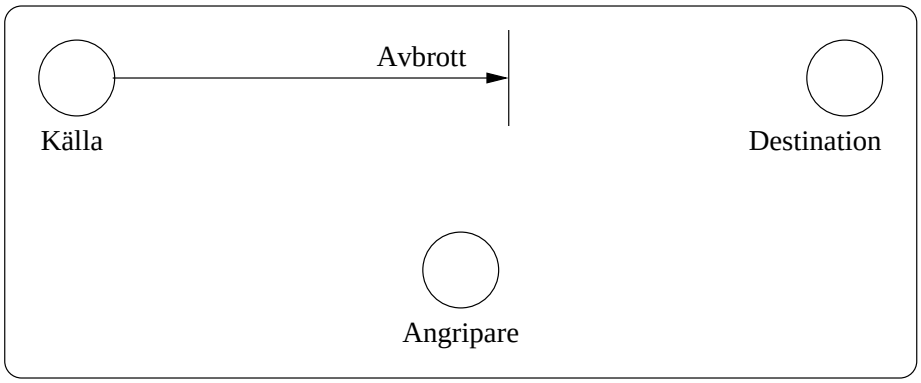
Både sekretess och integritet tappar sin mening om mottagaren inte kan verifiera att ett meddelande verkligen kommer från rätt avsändare. Ett givet meddelande kan snappas upp, ett nytt skapas, krypteras och checksummor beräknas. Om avsändaren inte är känd kan mottagaren inte lita på innehållet. Certifikat och privata/publika nyckelpar är vanliga sätt för att uppnå detta delmål. En attack som kan utföras om autentisering inte sker illustreras i figur 2.4.



Figur 2.4: Förfalskad kommunikation mellan två parter

Tillgänglighet

Ett attribut som inte kan uppnås genom att applicera någon specifik algoritm är tillgänglighet. För att uppnå önskad tillgänglighet krävs andra åtgärder. Tillgänglighet är dock beroende av ett stort antal faktorer och kan därför inte på ett enkelt sätt helt garanteras. En attack mot tillgängligheten illustreras i figur 2.5.



Figur 2.5: Avbruten kommunikation mellan två parter

För att utföra en attack riktad mot tillgängligheten behöver en angripare inte direkt tillgång till informationen som skickas. Detta gör att det blir en mera komplicerad process att skydda sig mot sådana attacker. Attacken kan utföras mot till exempel en router där meddelandet måste passera för att nå mottagaren. Detta kan ligga utanför den domän som sändare och mottagare har kontroll över. Att skydda meddelandet som skickas leder då inte till någon förbättring av tillgängligheten, då det ändå bara försvinner i tomma intet.

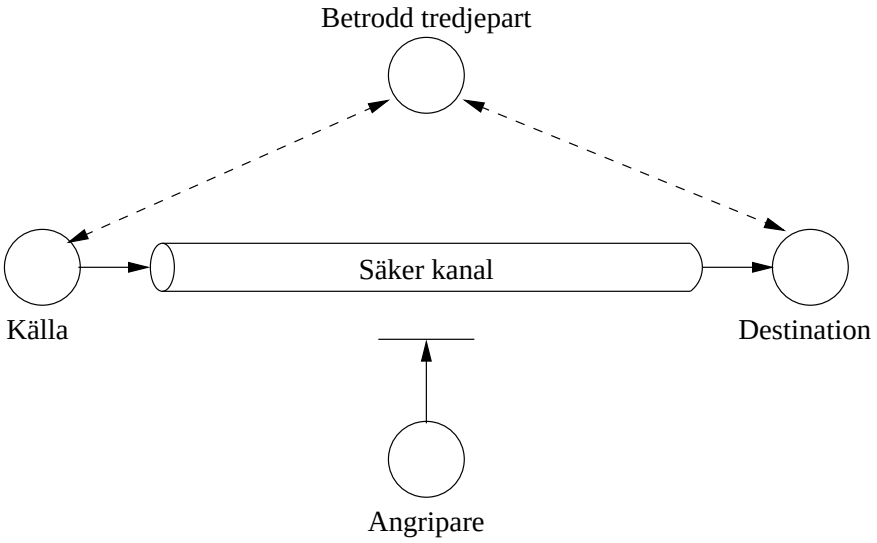
2.3.2 Säkerhetsmodell

Som vi har sett tidigare i detta kapitel kan ett meddelande skyddas med hjälp av algoritmer för kryptering och beräkning av checksummor. För att sändare och mottagare ska kunna autentisera varandra används certifikat eller privata/publika nyckelpar. För att etablera en säker kanal mellan sändare och mottagare kan en betrodd tredjepart blandas in

för autentisering och utbyte av hemlig information. Sådan information måste oftast delas mellan sändare och mottagare för de algoritmer som används vid kryptering. För att utbyta hemlig information kan även kryptering med privata/publika nyckelpar användas. För detta krävs ingen inblandning av tredje part.

När autentisering och utbyte av delad information mellan sändare och mottagare har skett kan en säker kanal mellan sändare och mottagare etableras. Uppstartsfasen vid skapandet av en sådan kanal refereras oftast till som handskakningsprocess. När en säker kanal etablerats mellan sändare och mottagare skickas meddelandet via denna och skyddas genom kryptering och beräkning av checksummor.

En generell modell som beskriver hur säker kommunikation kan uppnås mellan två parter via ett osäkert medium illustreras i figur 2.6.



Figur 2.6: Modell för säker kommunikation mellan två parter

Enligt modellen kan sekretess, integritet och autentisering kontrolleras av sändare och mottagare. Detta förhandlas fram i en så kallad handskakningsprocess. Meddelandet skyddas därefter med algoritmer för kryptering och beräkning av checksummor.

Kapitel 3

Konfigurerbar säkerhet

Konfigurerbar säkerhet är inte något som det talas om speciellt mycket, och de gånger det nämns i litteraturen diskuteras det om vilket krypto som ska användas till webbläsaren vid till exempel bankärenden eller om det ska användas 128- eller 168-bitars kryptering. Detta sker inte utan anledning. I dagsläget är det nämligen inte så mycket mer konfigurering som är möjlig när det gäller säkerhet. De val som erbjuds är nästan alltid binära; använd säkerhet eller använd inte säkerhet. Används mer avancerade applikationer finns valet mellan flertalet algoritmer som i praktiken gör samma sak och erbjuder samma typ av skydd. För en användare upplevs sällan någon större skillnad i prestanda heller.

Konceptet datasäkerhet i sig är fortfarande relativt nytt. Från att för bara några decennier sedan ha låst in stordatorn i ett rum med begränsad fysisk access och ingen möjlighet till fjärranslutningar har brandväggar och kryptering av nätverkstrafik fått ta över rollen som beskyddare av informationen.

Med konfigurerbar säkerhet menar vi ökade möjligheter när det gäller att välja säkerhetsnivå och sättet som denna nivå uppnås. Vi vill kunna ha mer val än att kryptera eller inte kryptera och vid kryptering välja 128- eller 168-bitars krypteringsnycklar. Vi anser att det inte alltid är nödvändigt att kryptera all data maximalt om säkerhet önskas, utan att det bör finnas mellannivåer på säkerhet. Dekryptering av data tar tid och prestanda, även

handdatorer med begränsad beräkningskapacitet och strömförsörjning eller arbetsstationer som inte har den senaste teknologin bör kunna ha ett visst mått av säkerhet i realtid.

En annan föråldrad tanke är att om inte en datamängd skyddas totalt är det ingen idé att skydda den alls. Att kryptera ett meddelande så starkt att det tar hundratals år att knäcka kryptot är inte nödvändigt om meddelandet tappar sitt värde efter en vecka. För en livesändning av ett evenemang över Internet kan det räcka att dekryptering av någon utan rätt nyckel inte sker i realtid för att det ska räknas vara tillräcklig säkerhet. När det gäller strömmande media i form av video eller musik kan det räcka att med hjälp av kryptering förstöra kvalitén minimalt för att en angripare inte ska anse det vara värt mödan. Vem skulle vilja se en film över Internet om strömmen är såpass krypterad att bara gråskalor och konturer framträder av filmen, eller lyssna på musik om det var femte sekund hörs ett högt och irriterande pipande ljud. Något som kommer att framgå i senare kapitel är att ju mer komplicerad struktur en datamängd har desto mindre ingrepp måste göras för att uppnå markanta skillnader i kvalitet.

I detta kapitel kommer några av de vanligaste protokollen och algoritmerna att presenteras och därefter undersöks om de är lämpliga för konfigurerbar säkerhet eller om någonting nytt måste skapas. Slutligen presenteras en metod för att uppnå konfigurerbar säkerhet. Detta koncept refererar vi till som partiell säkerhet.

3.1 Säkerhetsprotokoll

Många applikationer som kommunicerar över nätverk har i dagsläget implementerat egna säkerhetsmekanismer eller använder sig av externa moduler. Några exempel

på detta är Secure Shell (SSH) och Pretty Good Privacy (PGP) för att nämna några. Vi kommer dock inte att titta närmare på specifika applikationer då dessa oftast anpassar säkerheten beroende på vilka uppgifter de är tänkta för. För den som önskar läsa mer om PGP rekommenderas boken *PGP : Source Code and Internals* [Zim95], av Philip R.

Zimmermann, som är skaparen av PGP. Standardisering av de protokoll som SSH använder sig av har påbörjats. Detta har lett till ett antal Internet-Drafts, bland annat *SSH Transport Layer Protocol* [YKS+02].

Säkerhet har även implementerats på lägre nivåer i protokollstacken för att på så vis kunna hantera kommunikationen på ett mera generellt sätt. Exempel på detta är SSL/TLS och IPSec som vi ska titta lite närmare på i respektive underkapitel. För den som inte är bekant med Internets protokollstack visas en figur av denna i Appendix D. Ett ramverk för hur märkning av data för att hantera säkerheten i de olika lagren ska ske beskrivs i RFC 1457 [Hou93].

3.1.1 SSL/TLS

Secure Sockets Layer (SSL) är ett kommunikationsprotokoll som erbjuder en säker kanal mellan två maskiner. SSL arbetar på transportnivå i protokollstacken. Stöd finns inbyggt för att skydda datan och identifiera maskinen som kommunikationen sker mot. Den säkra kanalen är transparent vilket innebär att datan inte förändras när den passerar. Mottagaren kommer att läsa exakt samma data på sin sida som sändaren skickar. Transparensen innebär också att i princip alla protokoll som kan köras över TCP också kan köras över SSL.

SSL har genomgått flera förändringar sen den första versionen för allmänt bruk släpptes, till version 3.0 som används idag. SSL har också lett till utvecklingen av en ny standard, Transport Layer Security (TLS). Denna standard finns beskriven i RFC 2246 [DA99].

Alla versioner av SSL och TLS delar samma enkla utgångspunkt, att erbjuda en säker kanal mellan två kommunicerande applikationer över vilken godtycklig data kan skickas. En SSL-uppkoppling sker i flera steg där det absolut första är att skapa en TCP-uppkoppling mot servern. Utifrån denna TCP-socket byggs sedan SSL-lagret upp innan SSL-uppkopplingen slutligen sker. Vi ska här kortfattat beskriva SSL, för den som vill läsa mer rekommenderas boken *SSL and TLS: Designing and Building Secure Systems* [Res00], av Eric Rescorla.

Handskakningen

Innan data kan överföras sker en handskakning i SSL. Det finns tre syften med SSL-handskakningen. För det första måste klienten och servern komma överens om vilka algoritmer som ska användas för att skydda datan. För det andra måste de etablera de kryptografiska nycklar som ska användas av krypteringsalgoritmerna. För det tredje finns möjligheten för både klient och server att autentisera varandra. För verifiering används Message Authentication Codes (MAC).

MAC är en algoritm för beräkning av checksummor, som även tar med en nyckel i beräkningen. Så en MAC beror av både den nyckel som används och meddelandet som blir MAC:at. Beräkningen sker genom en funktion som tar en input av godtycklig längd och genererar en output av bestämd längd. Den viktigaste egenskapen hos algoritmen är att beräkningen bara går att utföra åt ett håll, en så kallad envägsalgoritm. Originalvärdet kan inte erhållas utifrån det beräknade värdet. Det är alltså inte möjligt att beräkna originalmeddelandet utifrån dess MAC. Eftersom en MAC bygger på en nyckel som är en delad hemlighet mellan sändare och mottagare är det inte möjligt för en angripare att beräkna en ny MAC om ett meddelande förändras på vägen.

Kortfattat kan handskakningen i SSL delas upp i fyra steg:

1. Klient och server utbyter information om protokollversion, vilka chiffer som stöds, session id, vilket chiffer klienten vill använda i första hand samt ett slumpnummer.

2. Servern kan skicka över ett certifikat, nyckelutbyte och begära certifikat av klienten. Servern signalerar efter detta slutet av "hello messagefasen.

3. Klienten skickar över sitt certifikat om det begärts. Klienten skickar sina nycklar och eventuellt verifikation på serverns certifikat.

4. Byte till utvald chiffer och slutet på handskakningsprocessen.

För nyckelutbyte stöds flera metoder; RSA, [Fixed, Ephemeral, Anonymous] Diffie-Hellman samt Fortezza.

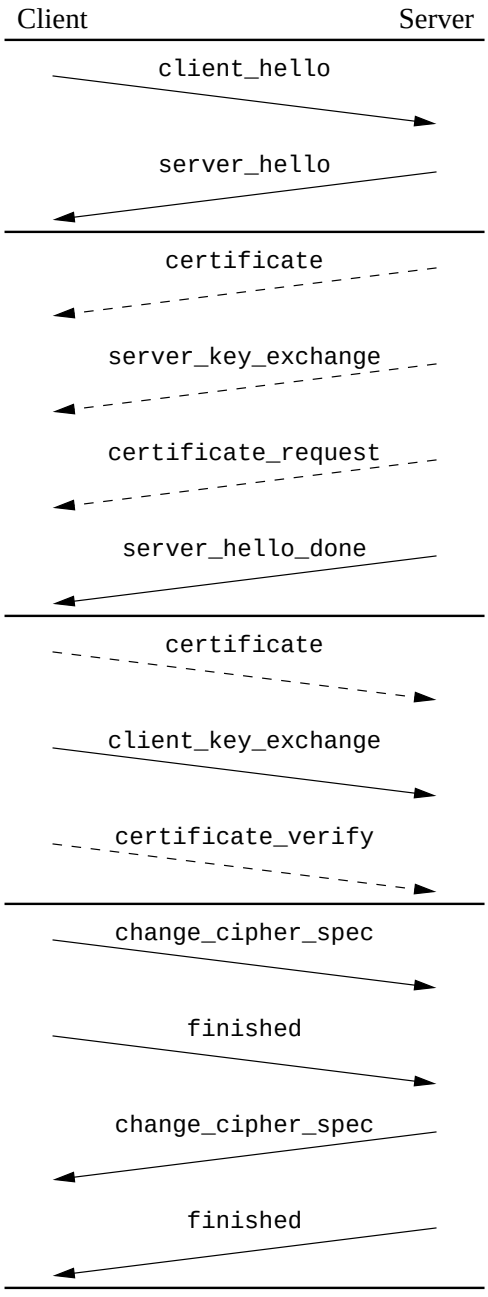
Vid ett nyckelutbytet skickas följande information:

- `ChipherAlgoritm`: Den algoritm som valts.
- `MACAlgoritm`: MD5 eller SHA-1.
- `ChipherType`: Strömmande eller blockorienterat.
- `IsExportable`: Sant eller falskt.
- `HashSize`: 0,16 (för MD5) eller 20 (för SHA-1) bytes.
- `Key Material`: En sekvens av bytes som innehåller data som använts i genereringen av nycklar.
- `IV Size`: Storleken av initieringsvärdena för Cipher Block Chaining (CBC) kryptering.

Efter en avslutad handskakning har servern och klienten kommit överens om vilka algoritmer som ska användas, vilka nycklar samt har gets möjlighet att verifiera varandra genom utbytet av certifikat. I slutet av denna process beräknas en checksumma på alla meddelanden som tagits emot. Denna checksumma skickas sedan över till den andra parten så att verifiering kan ske på att ingen har manipulerat handskakningen. I figur 3.1 visas ett exempel på handskakningsprocessen.

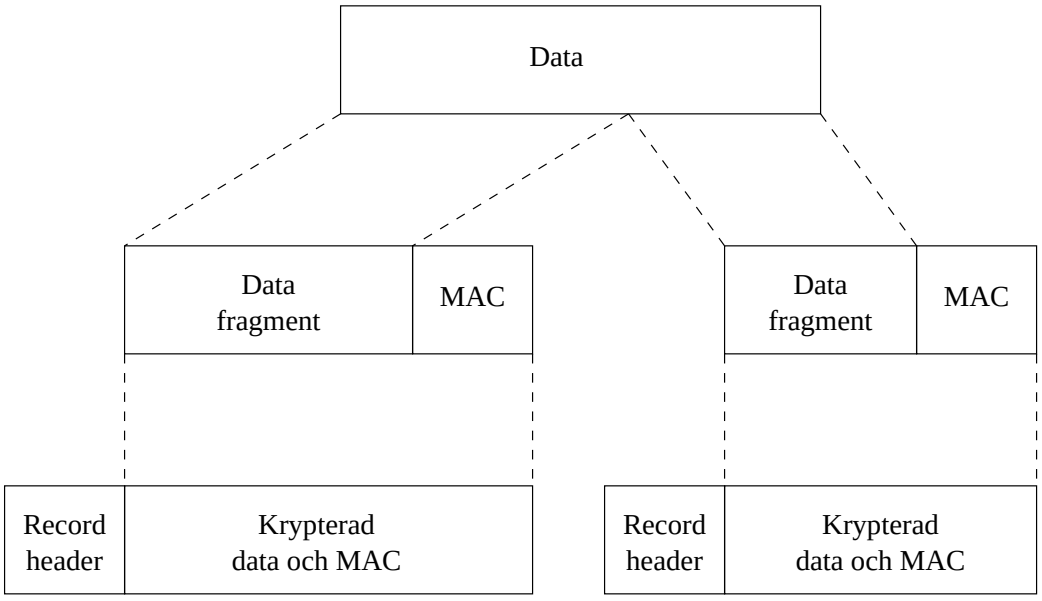
SSL Record Protocol

Det som åstadkommits än så länge är att parterna kommit överens om de algoritmer och nycklar som ska användas, samt att klienten verifierat serverns identitet med det certifikat som skickats. Att kunna dela på krypterad och autentiserad data ligger under SSL Record



Figur 3.1: SSL handshake

Protocols ansvar. Protokollet arbetar genom att dela upp dataströmmen som ska skickas i en serie fragment, vilka alla separat skyddas och sänds. Innan ett fragment kan skickas måste det skyddas. För att erbjuda integritet kan en MAC beräknas över fragmentet. MAC:en skickas tillsammans med datafragmentet och verifieras av mottagaren. MAC:en läggs på datafragmentet och krypteras till en så kallad encrypted payload. Slutligen läggs en header på och en record har skapats. Det är denna record som skickas till mottagaren. Detta illustreras i figur 3.2.



Figur 3.2: SSL data fragmentation

Record Header

Record Headers uppgift är att erbjuda den information som är nödvändig för att mottagaren ska kunna tolka den record som tagits emot. Denna information är meddelandetyp, längd och SSL-version. Längdfältet används så att mottagaren ska veta var meddelandet börjar. SSL versionen är en överflödig kontroll av att båda sidor är överens om versionen. Detta på grund av att version redan fastslagits i handskakningen. Meddelandetypfältet

identifierar vilken typ av meddelandet det är. I dagsläget finns följande fyra meddelandetyper:

1. `application_data`: All data som skickas av applikationer som använder sig av SSL faller under denna kategorin. Det är av denna typ alla meddelanden mellan två applikationer ovanför SSL är.

2. `alert`: Denna typen av meddelande används primärt för att signalera om olika typer av fel. Det vanligaste användningsområdet är att meddela vad som gått fel i handskakningen. Det andra användningsområdet är att meddela när uppkopplingen ska stängas.

3. `handshake`: Som namnet avslöjar är detta typen för alla meddelanden som skickas under handskakningsprocessen.

4. `change_cipher_spec`: Dessa meddelanden har en speciell innebörd. De indikerar ett byte av kryptonycklar. När handskakningsprotokollet har förhandlat fram nya nycklar skickas en `change_cipher_spec` för att meddela att dessa nya nycklar nu ska användas.

3.1.2 IPSec

IPSec erbjuder möjligheten för säker kommunikation över ett nätverk. Vanliga användningsområden är till exempel att skapa ett virtuellt privat nätverk över Internet, eller för fjärraccess från en enskild dator till ett lokalt nät. IPSec arbetar på nätverksnivån i protokollstacken och är därför transparent för de applikationer som använder sig av det. Konfiguration sker på systemnivå och är transparent för slutanvändaren. Systemet kan dock konfigureras för att passa individuella användares krav. RFC 1825 [Atk95] innehåller detaljerade specifikationer för den arkitektur som hanterar säkerhet på IP-nivå.

För att bestämma hur data ska hanteras i IPSec används Security Associations (SA). En SA beskriver en envägsrelation mellan sändare och mottagare. Dessa relationer beskriver hur säkerheten ska hanteras mellan sändare och mottagare. Två olika protokoll används för att erbjuda säkerhet. Båda dessa har stöd för två olika "modes" för hur data ska kapslas in. Skillnaden mellan dessa modes är hur IP-paketen byggs upp. De båda protokollen, samt de modes som kan användas beskrivs kortfattat i respektive underkapitel.

Authentication Header

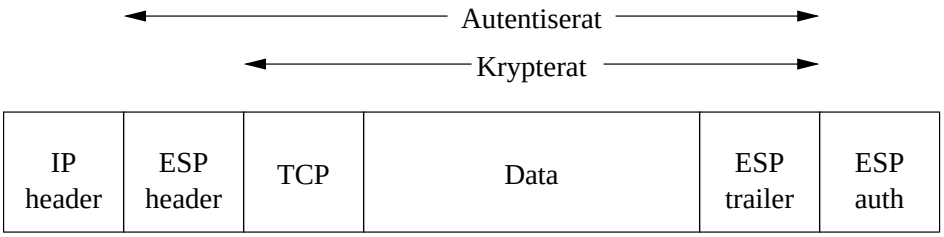
Authentication Header (AH) ger stöd för integritet och autentisering. AH försäkrar att om ett meddelande modifieras på vägen, kommer detta att upptäckas av mottagaren. Autentisering bygger på användandet av en MAC. Inget stöd för sekretess ges vid användande av AH.

Encapsulating Security Payload

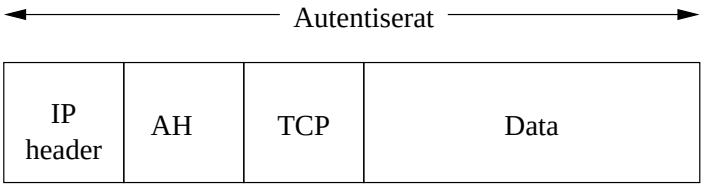
För att uppnå sekretess i IPSec används Encapsulating Security Payload (ESP). Med ESP krypteras meddelandets innehåll för att uppnå sekretess. ESP erbjuder även möjligheter till autentisering, vilket är valbart i och behöver inte utföras.

Transport mode

Transport mode erbjuder skydd för den data som skickas från de övre lagren. För transport mode behålls den vanliga headern och extra information läggs på efter denna. Kryptering och autentisering av IP-headern utförs inte när meddelanden skickas på detta sätt. Figur 3.3 visar vilka delar som krypteras och autentiseras för ett IPv4-paket i transport mode med ESP-protokollet. För AH sker ingen kryptering och de delar som autentiseras för ett IPv4-paket i transport mode med AH-protokollet illustreras i figur 3.4.



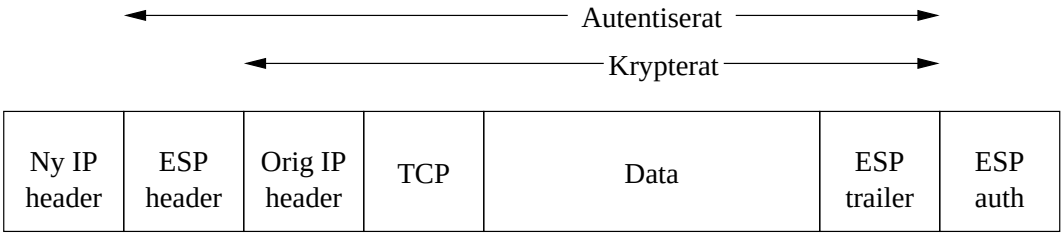
Figur 3.3: Paketstruktur för ESP-protokollet i transport mode



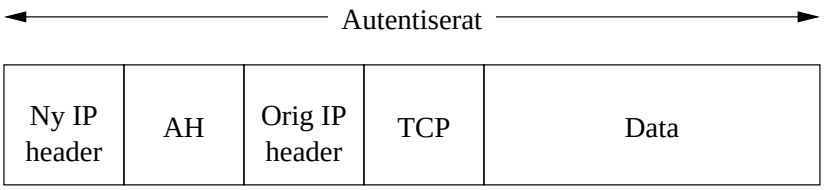
Figur 3.4: Paketstruktur för AH-protokollet i transport mode

Tunnel mode

Tunnel mode skapar en extra header och skickar med den ursprungliga headern som en del av IP-paketet. Detta leder till ökad säkerhet då autentisering och kryptering sker på hela det ursprungliga IP-paketet. Figur 3.5 visar vilka delar som krypteras och autentiseras för ett IPv4-paket i tunnel mode med ESP-protokollet. För AH-protokollet i tunnel mode autentiseras de delar av ett IPv4-paket som visas i figur 3.6.



Figur 3.5: Paketstruktur för ESP-protokollet i tunnel mode



Figur 3.6: Paketstruktur för AH-protokollet i tunnel mode

3.2 Begränsningar i befintliga säkerhetsprotokoll

Då multimediainnehåll blir allt mera vanligt förekommande på Internet bör protokoll som SSL/TLS ge möjlighet att anpassas till de prestandakrav som ställs för att hantera sådant innehåll. I både SSL och IPSec finns

möjligheten att hantera autentisering, verifiering och sekretess. De konfigurationsmöjligheter som finns beror på vilka algoritmer som finns implementerade i applikationen. Viktigast är att prestandan för de algoritmer som används vid kryptering och beräkning av checksummor kan påverkas då detta utförs i realtid på den data som skickas.

Prestandan behöver kunna anpassas så att om en mottagare med små resurser tar emot strömmande media som ska spelas upp i realtid bör mindre prestandakrävande algoritmer kunna väljas. I handskakningsprocessen sker en förhandling om vilka algoritmer som ska användas för kryptering och beräkning av checksummor. Vi ska i kapitel 3.3 gå igenom några av de krypteringsalgoritmer som används i dessa protokoll. Utifrån detta kan vi sedan se hur val av algoritmer påverkar säkerhet och prestanda. Vi koncentrerar oss på kryptering då detta är mera prestandakrävande än de algoritmer som beräknar checksummor.

IPSec som ligger på en lägre nivå i protokollstacken kan inte lika lätt anpassas till användarens önskemål. Konfigurering av säkerheten i IPSec sker via konfigurationsfiler och hanteras på ett statiskt sätt. Detta gör att omkonfigurering av säkerheten under drift blir omständig i IPSec. Trots detta har Christopher D. Agar skrivit sin master thesis om dynamisk parametrisering av IPSec [Aga01]. Målet med detta var att anpassa IPSec till de idéer om QoSS som arbetats fram av Irvine och Levine [IL00].

Till skillnad från IPSec har SSL möjligheten att när som helst göra en omförhandling av de algoritmer som används. Detta gör SSL bättre lämpat för att på ett dynamiskt sätt hantera säkerheten.

3.3 Krypteringsalgoritmer

Krypteringsalgoritmer kan delas in i två kategorier; symmetriska och asymmetriska algoritmer.

Asymmetriska algoritmer bygger på att kryptering och dekryptering inte sker med samma nycklar. På så sätt kan krypteringsnycklar distribueras publikt för att användas till kryptering av datan. Data kan sedan dekrypteras endast av den person som innehar den korrekta privata nyckeln. Asymmetriska algoritmer är dock långsamma och används i regel inte för att kryptera större datamängder.

I stället används symmetriska krypton där sändare och mottagare delar samma nyckel vilken används för både kryptering och dekryptering. Asymmetriska krypteringsalgoritmer används vanligtvis för distribution av de nycklar som används i symmetriska krypteringsalgoritmer. Vi ska i detta kapitel undersöka några av de symmetriska krypteringsalgoritmer som oftast används i implementationer av IPSec och TLS som diskuterades i kapitel 3.1.

Så gott som samtliga konventionella symmetriska kryptoalgoritmer bygger på en struktur känd som "Feistel Networks", vilket presenterades av Horst Feistel i artikeln *Cryptography and Computer Privacy* [Fei73]. Feistelnätverk är en kombination av substitution av värden och att förändra ordningen på dessa. För substitution används så kallade S-boxar som för varje värde som ges som input produceras ett nytt värde som output. För permutation används P-boxar för att skifta positionen mellan värdena. Meddelanden som ska krypteras delas först upp i block, sedan matas de in i S- och P-boxarna tillsammans med en nyckel. Att köra ett block genom boxarna för att erhålla ett krypterat block benämns som en runda i krypteringsalgoritmen. De algoritmer som använder sig av denna modell

körs i flera rundor för att på så vis erhålla högre säkerhet.

Vi koncentrerar oss här på DES och Trippel DES då detta är några av de vanligast förekommande algoritmerna. Några övriga vanligt förekommande algoritmer beskrivs också kortfattat. Prestanda för de beskrivna algoritmerna kommer sedan att jämföras och säkerheten i dessa algoritmer diskuteras.

3.3.1 Data Encryption Standard

Data Encryption Standard (DES) utvecklades av IBM under mitten av 70-talet och accepterades som en nationell standard i slutet av samma decennium. DES använder sig av ett symmetriskt nyckelblockchiffer, vilket bland annat innebär att samma nyckel används för både kryptering och dekryptering. Även om indatan som används som nyckel till DES är 64 bitar lång är nyckeln som faktiskt används av DES bara 56 bitar. Den minst signifikanta biten i varje byte är en paritetsbit, och sätts så att det totala antalet ettor i varje byte är udda. Dessa paritetsbitar ignoreras i krypteringsprocessen så att bara de sju mest signifikanta bitarna av varje byte används, vilket resulterar i en faktiskt nyckellängd på 56 bitar. Antalet rundor i DES-algoritmen är 16, blocklängden är 64 bitar och det finns stöd för flera olika varianter av kryptering. Dessa varianter finns för de flesta blockchiffer och kommer att beskrivas närmare i kapitel 3.3.4.

1998 lyckades Electronic Frontier Foundation, med hjälp av en specialbyggd dator som kallades för "DES cracker", att knäcka DES på mindre än tre dagar. Kostnaden för detta var under $250 000. Krypteringschipet som DES cracker använde hade en kapacitet som klarade av att kontrollera 88 miljarder nycklar per sekund. Tillvägagångssätt och resultat finns publicerat i bokform [Ele98].

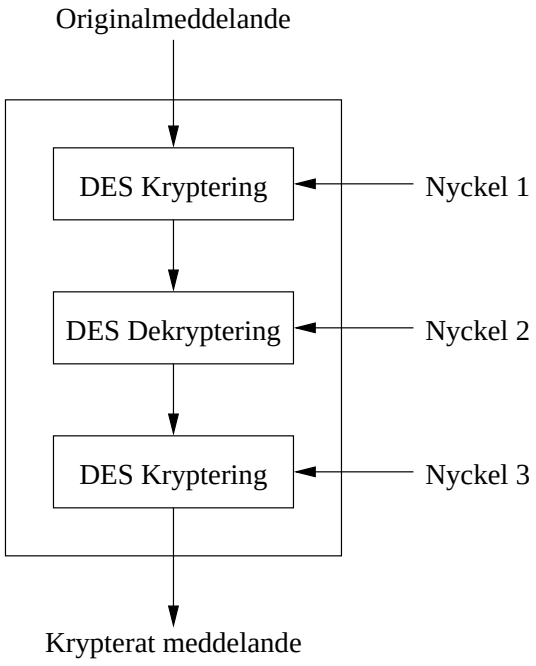
3.3.2 Trippel DES

Trippel DES (3DES), eller Trippel DEA som den också kallas, använder sig av samma algoritm som DES som beskrivs i kapitel 3.3.1. Den är tre gånger långsammare än DES,

såvida inte parallella beräkningar utförs, men förutsatt att den används rätt kan den vara enormt mycket säkrare. Anledningen till detta är att DES använder en 56-bitars nyckel medan 3DES använder en nyckel med en effektiv längd av 168 bitar.

3DES är enkelt sett en modifikation på DES som använder sig av tre stycken 64 bitars nycklar för en sammanlagt längd på 192 bitar. Det finns stöd för inmatning av en enda nyckel på 192 bitar (24 tecken) som sedan internt delas upp i tre delnycklar. På grund av paritetsbitarna är den faktiska nyckelstyrkan hos 3DES 168 bitar eftersom var och en av de tre delnycklarna innehåller åtta paritetsbitar som inte används under krypteringsprocessen.

Tiden för att kryptera en datamängd med en delnyckel är exakt samma som tiden att kryptera samma datamängd med DES algoritmen, men det upprepas tre gånger. Därav namnet 3DES. Datamängden krypteras först med nyckel ett, dekrypteras därefter med nyckel två för att slutligen krypteras igen med nyckel tre. Dessa krypteringssteg illustreras i figur 3.7.



Figur 3.7: Kryptering med 3DES

Följaktligen är 3DES tre gånger så långsam som DES, men är mycket säkrare om den används rätt. Att dekryptera något går till på samma sätt som när kryptering sker, förutom att allt då sker i omvänd ordning. Som dess föregångare DES använder sig 3DES av 64 bitars block av data för kryptering/dekryptering. Antalet rundor som utförs i 3DES blir tre gånger så många som i DES, detta innebär att antalet rundor för 3DES blir 48.

3.3.3 Andra vanliga krypteringsalgoritmer

IDEA

International Data Encryption Algorithm (IDEA), utvecklades av Xuejia Lai och James Massey år 1991 [LM91]. Nyckellängden i IDEA är 128 bitar och blockstorleken 64 bitar. Antalet rundor är endast åtta. En stor skillnad i rundorna jämfört mot DES är att IDEA inte använder sig av S-boxar. Substitutionen utförs i stället med en rad matematiska funktioner som gör den mycket svår att analysera.

Blowfish

Blowfish utvecklades 1993 av Bruce Schneier [Sch94]. Algoritmen utvecklades för att vara snabb och kompakt. Designen är dessutom gjord så att den ska vara lätt att implementera. Detta har lett till att Blowfish har blivit mycket populär. Då Blowfish är en relativt nyutveclad algoritm i jämförelse med DES så har inte lika mycket arbete lagts ner på att studera algoritmens svagheter, men ännu så länge har inga svagheter bevisats.

Blowfish använder sig på samma sätt som DES av S-boxar och XOR-operationer. Blowfish använder till skillnad från DES dynamiska S-boxar som genereras utifrån nyckeln. För att skapa S-boxar och delnycklar används själva Blowfishalgoritmen, som appliceras på nyckeln. För att skapa S-boxar och delnycklar exekveras algoritmen 521 gånger [Sta02], vilket gör algoritmen olämplig i de fall när nyckeln ofta byts.

Nyckellängden för Blowfish kan varieras upp till 448 bitar och prestandan för algoritmen

är inte beroende av nyckellängd. Vanligtvis används i praktiken ändå 128-bitars nycklar. Blockstorleken är precis som för DES 64 bitar.

RC5

RC5 utvecklades 1994 av Ron Rivest [Riv94]. RC5 finns definierad i RFC 2040 [BR96]. RC5 är en krypteringsalgoritm som tillåter att flera parametrar varieras. Då RC5 är ett wordorienterat krypto anpassas blockstorleken beroende på storleken av ett word i arkitekturen det används. Olika algoritmer används beroende på storleken av ett word. RC5 tillåter även varierad nyckelstorlek och antalet rundor som ska utföras.

3.3.4 Varianter av kryptering med 3DES

De flesta blockchiffer kan utföras på olika sätt när en hel fil krypteras. Förhållandet mellan de krypterade blocken kan hanteras på olika sätt. Vi ska här beskriva några av de krypteringslägen som kan användas i 3DES. Liknande konfigurationsmöjligheter finns hos de flesta blockchiffer.

Triple ECB

Electronic Code Book (ECB) är den vanligast använda metoden för DES och Triple ECB fungerar på exakt samma sätt. Datamängden delas in i block om 64 bitar och varje block krypteras för sig. Separat kryptering med olika block som är totalt oberoende av varandra har sina för- och nackdelar. Skulle datamängden sändas över ett nätverk eller telefonlinje kommer bitfel i sändningen endast att påverka det aktuella blocket. Det betyder också att blocken kan skiftas omkring och bytas ut mot varandra, dvs sortera om datamängden så att den inte kan kännas igen, utan att någon skulle märka det. ECB är den svagaste av de metoder som finns tillgängliga eftersom ingen extra säkerhet förutom DES-algoritmen är möjlig. Den är dock samtidigt den snabbaste och den som är enklast att implementera vilket gör den till det vanligaste valet till DES i kommersiella applikationer.

Triple CBC

Triple CBC (Cipher Block Chaining) är mycket närbesläktad med standard DES CBC mode. Varje block av ECB-krypterad chiffertext är XOR:ad med nästa okrypterade block som ska krypteras, vilket innebär att alla block har ett beroende till det föregående blocket. Detta betyder att för att kunna hitta den okrypterade texten måste nyckeln, samt chiffertexten för både nuvarande och föregående block vara kända. Det första blocket som krypteras har inget föregående block att XOR:as med, så den okrypterade texten XOR:as med ett 64-bitars nummer som kallas för initieringsvektor (IV). Så till skillnad från EBC kommer ett bitfel i dataöverföringen här att följa med framåt till alla efterliggande block eftersom varje block beror av det föregående. Denna metod är säkrare än ECB på grund av det extra XOR steget som lägger till ett extra lager med säkerhet. Som i Triple ECB är den effektiva nyckellängden 168 bitar och nycklarna används på samma sätt, men kedjeegenskaperna av CBC mode används också. Den första 64-bitars nyckeln används som IV till DES. Triple ECB exekveras sen för ett enda 64-bitars block av okrypterad text. Chiffertexten som blir resultatet av detta XOR:as med nästa block av okrypterad text. Proceduren upprepas sedan för varje 64-bitars block av text.

Triple CFB

Triple CFB (Cipher Feedback Mode) gör om blockchiffer till ett strömmande chiffer. Med ett strömmande chiffer är det möjligt att kryptera och dekryptera varje tecken i en ström direkt när de skickas. Dessutom produceras alltid så många tecken krypterad data som den data som skickas in. Detta gör att även om filen inte är en jämn multipel av blockstorleken behövs ingen utfyllnad, utan den krypterade datan blir lika stor som originaldatan.

För att uppnå detta används ett 64-bitars shiftregister. Som input till krypteringsfunktionen används en IV. De j mest vänsterställda bitarna i shiftregistret XOR:as med j bitar ur originalmeddelandet. Därefter skiftas registret j steg åt vänster och den producerade chiffertexten läggs till från höger i shiftregistret. För nästa del av originalmeddelandet

krypteras shiftregistret och proceduren upprepas.

3.3.5 Säkerhet i befintliga krypteringsalgoritmer

I rapporten *Minimal Key Length for Symmetric Ciphers to Provide Adequate Commercial Security* [BDR+96] skriven av Blaze med flera räknas en krypteringsalgoritm som stark om följande villkor är uppfyllda:

1. Det finns inga genvägar till att återställa originalmeddelandet utom att använda "brute force" för att testa nycklar tills den rätta nyckeln hittas.

2. Antalet möjliga nycklar ska vara tillräckligt stort för att göra en sådan attack tillräckligt svår för att inte vara praktiskt genomförbar.

Inga kända genvägar finns i de algoritmer som beskrivits tidigare i detta kapitel, så i dagsläget får samtliga av de beskrivna algoritmerna anses vara starka. Det har dock visats för DES-algoritmen att på grund av dess nyckellängd kan den inte längre räknas som en säker krypteringsalgoritm. Detta beskrevs i kapitel 3.3.1.

Att en krypteringsalgoritm anses vara stark innebär alltså inte att den är helt oknäckbar, utan att det inte finns några kända genvägar till hur den ska knäckas. Denna klassificering av säkerhet är ett dock ett binärt sätt att beskriva säkerheten. Detta tar ingen hänsyn till värdet av den data som ska skyddas, utan delar endast upp krypteringsalgoritmer i starka och svaga algoritmer. Vi inför därför också begreppet "beräkningssäker" som i Stallings bok *Network Security Essentials* [Sta02] definieras på följande sätt, där algoritmen måste uppfylla åtminstone ett av följande krav:

1. Kostnaden för att knäcka chiffret överskrider värdet av den krypterade informationen.

2. Tiden det tar att knäcka chiffret överstiger tiden som den krypterade informationen är användbar.

Detta inför parametrar som tid och pengar för att avgöra om en algoritm antas vara tillräckligt säker. I detta fall är säkerhetsnivån som krävs beroende av

värdet på informationen som ska skickas.

3.3.6 Prestanda för befintliga krypteringsalgoritmer

För att få en uppfattning om prestandan för olika krypteringsalgoritmer kan antalet klockcykler som krävs för kryptering av en byte data vara ett bra mätvärde. Tabell 3.1 är en sammanställning av antalet klockcykler som krävs för några vanliga krypteringsalgoritmer. Tabellen är hämtad från Counterpane Labs hemsida [Sec03].

Algoritm	Klockcykler per runda	Antal rundor	Klockcykler per krypterad byte
Blowfish	9	16	18
RC5	12	16	23
DES	18	16	45
IDEA	50	8	50
3DES	18	48	108

Tabell 3.1: Hastighetsjämförelser av blockchiffer

3.3.7 Problem med användandet av befintliga algoritmer

För att konfigurerbar säkerhet ska kunna fungera på ett tillfredsställande sätt räcker det inte att kunna välja vilken krypteringsalgoritm som används. De algoritmer som används i dagsläget är alla så kallade säkra algoritmer, vilket definierades i kapitel 3.3.5.

Då multimedia oftast inte har ett allt för högt ekonomiskt värde så är säkerheten de konventionella algoritmerna erbjuder tilltagen i överkant. Detta då det är ekonomiskt oförsvarbart att spendera en stor summa pengar på att knäcka en videoström, när samma film kan hyras i en videobutik betydligt billigare. Så det vore önskvärt om säkerheten kunde minskas till förmån för ökad prestanda i ett sådant fall.

Detta innebär att nya vägar måste utforskas och kontroversiella idéer tillåtas utrymme. En av dessa nya idéer leder oss vidare till partiell säkerhet som kommer att presenteras i kapitel 3.4. Den partiella säkerheten tillåter att säkerheten kan minskas successivt till förmån för prestanda.

3.4 Partiell säkerhet

För multimediaapplikationer som kräver en viss nivå av säkerhet samtidigt som prestandakrav ställs för att kunna spela upp strömmande media i realtid är det viktigt att kunna sänka säkerhetsnivån tillräckligt för att motsvara de prestandakrav applikationen har i övrigt. Vi gav i kapitel 3.3.7 ett exempel på ett fall där detta är önskvärt.

Om innehållet är av en sådan karaktär att det räcker att införa brus för att innehållet helt tappar sitt ekonomiska värde kan en metod för kryptering vara att kryptera endast en delmängd av datan. Genom att kryptera endast delar av datamängden införs ett brus och beroende på hur högt brus som tolereras kan andelen krypterad data varieras. Samma metod skulle kunna genomföras för integritetskontroll. Om mottagaren kan försäkra sig om att en del av paketen kommer från den angivna avsändaren kan detta vara tillräckligt.

Vi ska vidare i detta kapitel beskriva konceptet med partiell säkerhet, genom att ta upp några av de möjligheter som finns med detta genom partiell kryptering och integritetskontroll.

3.4.1 Partiell kryptering

Som en del av konfigurerbar säkerhet kan partiell kryptering vara en möjlighet att variera säkerheten. Med partiell kryptering menar vi i detta fall att endast vissa paket i dataströmmen krypteras, eller att endast vissa delar av informationen krypteras.

Enklaste fallet av partiell kryptering vore att låta användaren välja hur stor del av datamängden som ska krypteras, för att sedan jämnt distribuera krypteringen över data-

mängden. I vissa fall skulle förbättringar för att höja säkerheten vara möjliga beroende av vilken trafik som skickas. För en videoström skulle kryptering av endast I-frames göra hela videoströmmen oanvändbar såvida inte arbete läggs ner på att dekoda dessa frames, alterativt ta bort dem och se på en allvarligt försämrad bild. I-frames är den delen av en videoström där mesta bildinformationen finns lagrad. Övriga frames beskriver förändringar i bilden. Således kan delar av videoströmmen fortfarande urskiljas utan I-frames men bildkvalitén kommer då vara allvarligt försämrad. Detta kan i flera fall vara nog för att en angripare ska ge upp. Några sådana försök har tidigare genomförts. Vi ska senare i detta kapitel titta på kryptering av videoströmmar.

Ytterligare möjligheter för partiell kryptering skulle kunna vara att en grundström skickas okrypterad, medan tilläggsinformation skickas krypterad. Detta skulle kunna användas för att visa en lågupplöst videoström som i kombination med tilläggsinformationen ger en högre kvalitet på videoströmmen. Detta skulle medföra att ett företag skulle kunna erbjuda en gratisversion av någon livesändning och sedan mot betalning erbjuda ökad bildkvalitet.

Genom att utföra partiell kryptering av dataströmmen medför det att krypteringsnivån direkt får ett mätbart värde, från helt okrypterad till helt krypterad. Dessa värden är dessutom helt oberoende av vilken algoritm som används för krypteringen. Då det finns en mängd krypteringsalgoritmer som används i dagsläget som får anses vara säkra, vilket konstaterades i kapitel 3.3.5, kan partiell kryptering vara ett sätt för användaren att välja krypteringsnivå.

Säkerheten kommer dock att bli beroende av vilken typ av data som skickas. Genom att kryptera I-frames för en videoström görs hela strömmen oanvändbar om den inte dekrypteras. För ett klartextmeddelande som delvis krypteras kan de okrypterade delarna fortfarande läsas av obehöriga. I många fall kan det finnas ett starkt databeroende så att de krypterade delarna av datan kan återskapas utan att behöva dekrypteras om tillräckligt mycket information lämnas okrypterad.

Kryptering av bilder

Vi ska i kapitel 4 närmare undersöka vilka möjligheter partiell kryptering kan medföra för bilder. Försök med partiell kryptering av bilder har tidigare utförts, bland annat i en artikel av Droogenbroeck och Benedett [DB02].

Kryptering av video

Thomas Kunkelmann och Uwe Horn har skrivit en artikel som jämför olika typer av optimering för kryptering av video [KH98]. Denna artikel tar upp flera av de försök som har utförts för att optimera krypteringen av video. Dessa försök kan alla klassas som partiell kryptering. Vissa delar av filen väljs ut och genom att kryptera dessa erhålls önskad krypteringsnivå.

Kryptering av exekverbara filer

För exekverbara filer kan en mycket låg andel av kryptering vara ett tillräckligt skydd. Oftast blir en exekverbar fil obrukbar vid enstaka bitfel. Utan att knäcka krypteringen kan de krypterade blocken vara mycket svåra att återskapa.

3.4.2 Partiell integritetskontroll

Integritetskontroll är inte beroende av filens innehåll på samma sätt som kryptering. Integritetskontrollen kan spridas ut på delar av filen för att på så sätt få en skala för säkerheten.

Det enda som behövs för att införa partiell integritetskontroll i en applikation som redan använder sig av full integritetskontroll är möjligheten för mottagaren att välja hur stor andel av datamängden som ska verifieras.

Oftast sker verifiering på varje paket i en dataström. I detta fall räcker det att mottagaren väljer ut vilka paket som önskas verifieras. Ibland kan verifiering dock ske på hela

meddelandet och då krävs en förhandling mellan sändare och mottagare om vilken datamängd som ska verifieras.

Kapitel 4

Tester med partiell kryptering

För att få en visuell uppfattning om hur data påverkas vid partiell kryptering har vi valt att koncentrera oss på kryptering av bilder. Vi har använt oss av en bild i TIFF-format som utgångspunkt, då detta format är okomprimerat. Denna bild konverterades sedan till JPEG-format för fortsatta tester.

Eftersom JPEG-standarden erbjuder en mängd olika möjligheter för bildkomprimering kommer vi till att börja med beskriva JPEG-formatet och de kodningsstandarder som används för att skapa JPEG-bilder.

Då resultatet vid partiell kryptering beror på formatet av den data som krypteras kommer vi i kapitel 4.4 att utveckla en metod för partiell kryptering som blir oberoende av vilket filformat som används vid krypteringen.

4.1 JPEG-kodning

För att se vad som händer med en bild när den komprimeras till JPEG-format ska vi i detta kapitel beskriva de steg som utförs vid kodningen av en JPEG-bild. Det exempel vi beskriver är en sammanfattning av den beskrivning till JPEG-kodning som ges i boken *Multimedia Communications* av Fred Halsall [Hal00]. Detta exempel beskriver hur

kodningen sker för en bild i baseline mode. Kodningen sker i följande fem huvudsteg:

1. Bildförberedelse
2. Forward DCT
3. Kvantisering
4. Entropikodning
5. Ramkonstruktion

Varje steg beskrivs separat i respektive underkapitel.

4.1.1 Bildförberedelse

Beroende på originalbildens format presenteras bilddata på olika sätt. Ett vanligt sätt att representera bilder är RGB-formatet, där tre matriser används för att beskriva värdet för färgerna röd, grön och blå. Alternativt kan bilderna beskrivas med krominans (C_b och C_r) och luminans (Y), där krominans beskriver färginformation och luminans beskriver ljusstyrka. I detta fall kan krominansvärdena C_b och C_r oftast minskas i storlek utan märkbar bildförsämring. Varje matris hanteras separat i kodningen. Innan DCT-beräkningarna delas matriserna upp i block, till mindre 8x8-matriser. Detta för att beräkningarna är lättare att genomföra på mindre matriser.

4.1.2 Forward DCT

Efter att matriserna delats upp i block används Discrete Cosine Transform (DCT), där pixelvärdena görs om till komponenter av spatiala frekvenser, där varje komponent beskriver styrkan för en viss frekvens. Utifrån beräkningarna skapas en ny matris där den horisontella frekvensen ökar åt höger och den vertikala frekvensen ökar nedåt i matrisen. Det värde som ligger längst upp till vänster i matrisen kallas för DC-koefficient och är ett medelvärde av alla 64 värden från ursprungsmatrisen.

4.1.3 Kvantisering

Kvantiseringen är den del av JPEG-kodningen där den största delen av informationen går förlorad och bildförstöringen sker. Eftersom det mänskliga ögat inte är lika känsligt för höga frekvenser kan de högre frekvenserna nollställas utan att bilden märkbart försämras. För att avgöra hur detta ska modifieras används en så kallad kvantiseringstabell för att avgöra vilka frekvenser som ska modifieras. Kvantiseringen utförs på så sätt att varje koefficient divideras med ett tröskelvärde och avrundas. Vid avkodningen multipliceras koefficienten med samma värde. För de högre frekvenserna används högre tröskelvärden.

4.1.4 Entropikodning

Efter kvantiseringen används ett antal metoder för att komprimera bilddatan. Ingen av dessa är bildförstörande. Varje komprimeringsalgoritm beskrivs i respektive underkapitel.

Vektorisering

De algoritmer som används opererar på endimensionella vektorer. För att skapa vektorer från de matriser som erhållits utförs vektorisering på dessa. Vektoriseringen sker i ett zig-zag-mönster. Inläsningen startar i DC-koefficienten och sker i ett zig-zag-mönster genom matrisen så att de koefficienter med lägst frekvens hamnar först i vektorn och de högsta sist. Kodningen av DC-koefficienter sker sedan separat från övriga komponenter i vektorn. DC-koefficienterna för samtliga block kodas med differentialkodning, medan övriga koefficienter kodas med run-lengthkodning för varje block.

Differential encoding

Eftersom storleken av blocken inte är så stor skiljer sig inte DC-koefficienten så mycket från ett block till nästa. Värdena för DC-koefficienterna sätts därför som värden relativa till det föregående. Den första koefficienten får behålla sitt ursprungsvärde, sedan beräknas

värdet på efterföljande koefficienter som skillnaden mot föregående värde.

Run-length encoding

Tack vare zig-zagscanningen innehåller vektorn långa sekvenser av nollor. Detta gör att run-length encoding lämpar sig väl för att komprimera denna. Data sparas i par, där ena värdet är antalet nollor till nästa värde och det andra är värdet för nästföljande koefficient.

Huffman encoding

Slutligen komprimeras all bilddata med huffmankodning, vilket gör att all bilddata byts ut mot kodord av variabel storlek. De mest frekventa värdena

tilldelas kortare kodord och på detta sätt minskas datamängden.

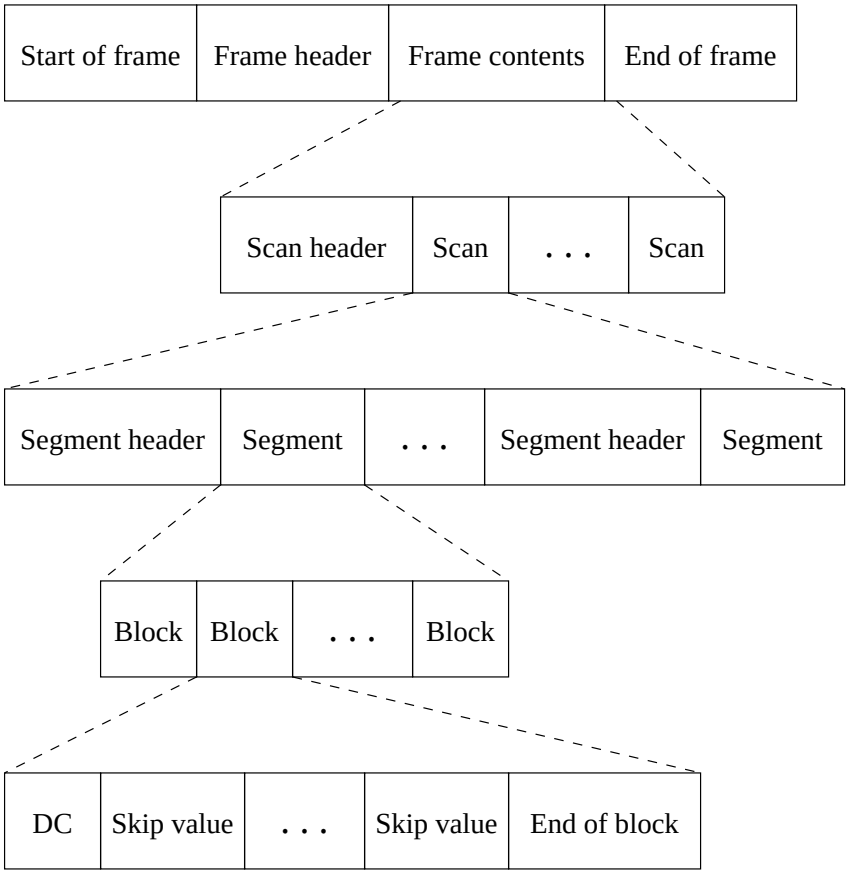
4.1.5 Ramkonstruktion

För att bilden ska kunna avkodas av en mottagare måste denna kunna tolka värdena i bitströmmen. JPEG-standarden definierar därför strukturen för den totala bitströmmen. Vi ska i kapitel 4.2 beskriva strukturen för en JPEG-bild mera ingående för att se vilken information filen innehåller förutom den komprimerade bilddatan.

4.2 Filstruktur för en JPEG-bild

JPEG-data består av två klasser av data; entropikodade segment och headerdata. Entropikodade segment innehåller den entropikodade datan, medan headerdata innehåller information om hur bilden ska tolkas och avkodas.

Figur 4.1 illustrerar hur data är organiserad i en JPEG-fil. Varje segment i figuren innehåller entropikodad bilddata, medan övriga block innehåller markörer med information om hur bilden är uppbyggd.



Figur 4.1: Hur data är strukturerad en JPEG-bild

Den kodade JPEG-strömmen innehåller markörer som visar hur data är organiserad i filen.Varje markör börjar med värdet FF hexadecimalt och följs av en markörkod med en bytes storlek. De markörkoder som förekommer i JPEG-strömmen listas i tabell 4.1 och 4.2. Dessa tabeller har hämtats ur boken *JPEG Still Image Data Compression Standard* [PM93].

Namn	Kod	Längd	Kategori
SOF_0	FFC0	Variabel	Baseline DCT
SOF_1	FFC1	Variabel	Extended sequential DCT
SOF_2	FFC2	Variabel	Progressive DCT
SOF_3	FFC3	Variabel	Lossless (sequential)
SOF_5	FFC5	Variabel	Differential sequential DCT
SOF_6	FFC6	Variabel	Differential progressive DCT
SOF_7	FFC7	Variabel	Differential lossless
SOF_9	FFC9	Variabel	Extended sequential DCT
SOF_{10}	FFCA	Variabel	Progressive DCT
SOF_{11}	FFCB	Variabel	Lossless (sequential)
SOF_{13}	FFCD	Variabel	Differential sequential DCT
SOF_{14}	FFCE	Variabel	Differential progressive DCT
SOF_{15}	FFCF	Variabel	Differential lossless

Tabell 4.1: Start Of Frame markörer i JPEG-strömmen

4.2.1 Headerdata

Den headerdata som återfinns i en JPEG-bild inleds alltid med någon av de kända markörer som listats i tabell 4.2. Denna markör specificerar vilken bildtyp det är. Därefter följer information om kodtabeller och liknande. Oftast kan kommentarer förekomma i inledningen av en fil. Till exempel bilder skapade i Photoshop innehåller kommentarer om att de är skapade i just detta program.

Varje scan inleds också med en header som specificerar hur denna scan är kodad. Vanlig information här kan vara vilken huffmantabell som använts för just den scannen om flera huffmantabeller specificerats för filen.

Namn	Kod	Längd	Kategori
APP_n	FFE0 - FFEF	Variabel	Applikationsspecifik
COM	FFFE	Variabel	Kommentar
DAC	FFCC	Variabel	Aritmetisk kodtabell
DHP	FFDE	Variabel	Hierarkisk progression
DHT	FFC4	Variabel	Huffmantabell(er)
DNL	FFDC	4	Antal rader
DQT	FFDB	Variabel	Kvantiseringstabell(er)
DRI	FFDD	4	Omstartsintervall
EOI	FFD9	0	Bildslut
EXP	FFDF	3	Expandera referensbilder
JPG	FFC8	Odefinierad	Reserverad för utökningar
JPG_n	FFF0 - FFFD	Odefinierad	Reserverad för utökningar
RES	FF02 - FFBF	Odefinierad	Reserverad
RST_m	FFD0 - FFD7	0	Restart
SOI	FFD8	0	Bildstart
SOS	FFDA	Variabel	Start av scan
TEM	FF01	0	Temporär aritmetisk kodning

Tabell 4.2: Övriga markörer i JPEG-strömmen

4.2.2 Bilddata

Markören SOS talar om var i filen bilddata börjar. Beroende på bildtyp kan en bild innehålla en eller flera SOS. Inne i ett block med bilddata kan RST-markörer förekomma. För att behålla JPEG-formatet kan inte dessa krypteras. Bilddata med värdet FF måste också följas av 00, vilket är en utfyllnad för att skilja värdet FF från en ny markör.

Entropikodad data i en JPEG-ström består av en eller flera ramar. Varje ram består av en eller flera scanningar av bilddatan. En scanning går igenom bilddata för en eller flera komponenter av bilddatan. Innan varje ram finns ett markörsegment som innehåller parametrar som behövs för avkodningen av scanningen.

Den entropikodade datan i varje ram kan delas upp i flera segment. Uppdelningen sker genom att speciella omstartsmarkörer placeras i den entropikodade datan. Dessa markörer kan urskiljas i dataströmmen utan att datan avkodas. Avkodning av datan sker individuellt

för varje segment av entropikodad data i strömmen.

4.3 Partiell kryptering av bilder

Vi har använt oss av en bild av en modell vid namn Lena för att visa vilka resultat som kan erhållas med partiell kryptering. Den originalbild vi använde oss av visas i figur 4.2.

Figur 4.2: Originalbilden av Lena i TIFF-format

Originalbilden vi använde oss av är i TIFF-format. Bildinformationen för denna har listats i appendix A.1.

4.3.1 Partiell kryptering av TIFF-bild

Det första testet som utfördes var att förändra delar av bildinformationen i originalbilden. För att utföra dessa förändringar användes ett program vi skrivit som byter ut valda bytes i en fil mot slumpmässiga värden. Källkoden för detta program finns listad i appendix B.1.

För det första försöket valdes en krypteringsgrad där var 16:e byte av bilddatan krypterades. Krypteringen skedde inte med en riktig krypteringsalgoritm, utan simulerades genom att byta ut data mot slumpmässigt vald data. Resultatet av det ursprungliga försöket visas i figur 4.3

Figur 4.3: Kryptering av var 16:e byte i TIFF-bilden

Då detta inte visade sig ge så stor förändring i utseendet av bilden utfördes försöket flera gånger med högre antal krypterade block. Detta utfördes med en krypteringsgrad där mellan 12,5% och 75% av bilddatan krypterades. Resultatet vid ökad krypteringsgrad visas i figur 4.4.

4.3.2 Partiell kryptering av JPEG-bild

Originalbilden konverterades även till JPEG-format, vilket är vanligt förekommande för bilder som distribueras via nätverk. Detta beror på att JPEG-komprimering oftast är ett mycket effektivt sätt att minska filstorleken av bilder, utan att bildkvaliteten försämras

Kryptering av var 8:e byte

Kryptering av var 4:e byte

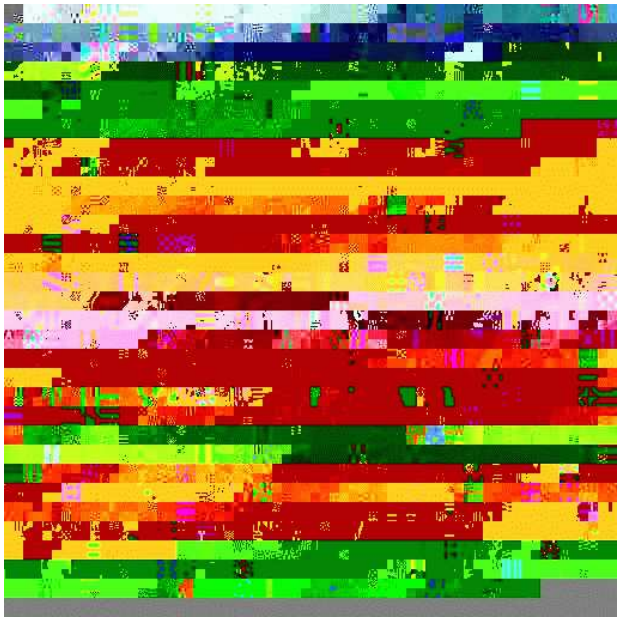
Kryptering av var 2:a byte

Kryptering av 3 av 4 bytes

Figur 4.4: Högre krypteringsgrad i TIFF-bilden

nämnvärt. Beskrivning om hur bilden komprimerades till JPEG-format samt bildinformation finns listat i appendix A.2.

För att bilden ska vara möjlig att titta på valde vi att endast kryptera bilddata, medan headerdata lämnades intakt. Vi valde även att inte generera värdet FF för de krypterade blocken, då detta indikerar att det kan vara en markör om det inte efterföljs av värdet 00. På samma sätt som för testerna på TIFF-bilder som beskrevs i kapitel 4.3.1 valde vi att utgå från kryptering av var 16:e block. Resultatet för JPEG-bilden visas i figur 4.5.

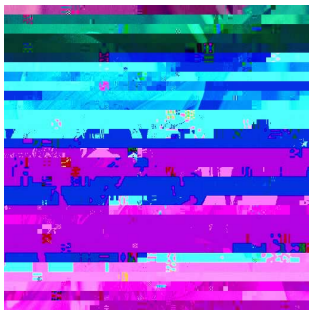


Figur 4.5: Kryptering av var 16:e byte i JPEG-bilden

För JPEG-bilden visade sig resultatet vara betydligt bättre än för TIFF-bilden. Vi valde därför att upprepa försöket med minskad krypteringsgrad. Försöken upprepades med krypteringsgrader ner till så lite som var 512:e byte krypterades. Resultatet visas i figur 4.6.

Detta har visat sig ge väldigt bra resultat för filens utseende, även vid låg andel krypterad data. Vad som händer är att bilddata blir osynkronicerad mot det huffmanträd som används vid avkodningen och hela bilden förvrängs. Detta är dock möjligt att återsynkronicera och på så vis återskapa delar av bilden. Vi ska i kapitel 4.5 närmare beskriva

Kryptering av var 64:e byte



Kryptering av var 128:e byte

Kryptering av var 256:e byte

Kryptering av var 512:e byte

Figur 4.6: Lägre krypteringsgrad i JPEG-bilden

JPEG-formatet och titta på de möjligheter som erbjuds att kontrollera vilka delar som ska krypteras för att hålla säkerheten på en hög nivå.

4.3.3 Svagheter i partiell kryptering

Partiell kryptering har visat sig ge varierat resultat beroende på vilken data som krypteras. Detta medför att andelen krypterad data inte ger ett mått på säkerheten i krypteringen, utan endast blir ett mått på prestandaförändring. Vi ska därför i kapitel 4.4 undersöka möjligheten att utveckla en generell metod för kryptering för att erhålla ett resultat som liknar det för JPEG-bilder oavsett filformat. Det vill säga att trots att endast vissa delar av filen krypterats, ska övriga delar inte direkt kunna identifieras.

Dock har de tester vi utfört med partiell kryptering av JPEG-bilder visat lovande resultat och vi ska i kapitel 4.5 undersöka möjligheten att utveckla idén om partiell kryptering i de fall där filformatet är känt.

4.4 Lättviktskryptering

De tester som utfördes med partiell kryptering i kapitel 4.3 visade att säkerheten som erhålls vid partiell kryptering varierar beroende på formatet av den data som krypteras. Det visade sig att partiell kryptering lämpade sig bättre för JPEG-bilder i jämförelse med TIFF-bilder. Detta beror på att bilddata i JPEG-bilder är komprimerad och vid kryptering av enskilda block kommer även övriga block att påverkas genom att data i dessa blir osynkronicerad.

För att kunna återskapa de okrypterade delarna av JPEG-bilden så krävs att återsynkronicering sker mot det huffmanträd som använts för att koda bilddata. Säkerheten i detta fall blir beroende av hur svårt det är att återsynkronicera data mot huffmanträdet.

För att uppnå en liknande effekt för vilket filformat som helst kan ett extra "krypteringssteg" utföras på hela filen innan partiell kryptering sker. Kravet för detta är att det ska förändra datan på ett sådant sätt att originaldatan inte längre direkt kan utläsas. Det

ska sedan kunna återskapas av mottagaren. Det får inte heller vara resurskrävande att applicera. Prestandan får inte försämras till den grad att det är mera resurskrävande än att applicera en befintlig krypteringsalgoritm på filen.

Vi behöver en algoritm som kan appliceras på en fil som förändrar filens innehåll så att originaldatan inte kan utläsas ur den förändrade filen. Följande krav ställs på algoritmen:

1. All data i filen ska förändras så att inte originaldata kan utläsas direkt ur filen utan att den återskapas.

2. Mottagaren ska kunna återskapa data från den förändrade filen tillbaka till ursprungsdata utan dataförluster.

3. Algoritmen ska vara mindre tidskrävande än konventionella krypteringsalgoritmer att utföra.

Vad som skiljer kraven vi ställer på algoritmen från de krav som ställs på en stark krypteringsalgoritm som beskrevs i kapitel 3.3.5 är att genvägar till att kunna återskapa originaldata tillåts. I detta läge sätter vi inte heller några begränsningar på nyckellängden. Vi kommer dock i kapitel 4.4.3 diskutera detta ytterligare.

4.4.1 Kryptering med XOR

XOR, eller exklusivt OR är en bitvis boolesk operation som tar in två bitar och producerar en bit som output.

Sanningstabell för XOR kan ses i tabell 4.3. Den första kolumnen bör ses som ren okrypterad data, den andra som nyckeln och den tredje som det värde XOR-operationen producerar.

Som kan ses i tabell 4.3 kan varje bit okrypterad data representeras av två olika bitar krypterad data. Vid ett krypterat värde på '0' är det omöjligt att med säkerhet veta vilket

A	B	A $\oplus$ B
0	0	0
0	1	1
1	0	1
1	1	0

Tabell 4.3: Sanningstabell för XOR

värde som fanns från början eftersom det både '1' och '0' kan producera värdet '0' som output. Det är detta som leder till att det för varje XOR:ad bit finns 2 möjliga grundvärden.

Utan att känna till vilken nyckel som använts för att producera det krypterade värdet är det alltså omöjligt att känna till originalvärdet. Detta uppfyller det första kravet vi ställde på en algoritm i början av detta kapitel. Vi ska i kapitel 4.4.3 närmare titta på säkerheten i XOR-operationen.

För att bitvis utföra XOR för att kryptera en fil krävs en nyckel till krypteringen som är lika stor som den datamängd som ska krypteras. Detta leder till en orealistiskt hög overhead för mängden data som måste skickas. Kryptering av en fil kan därför i stället utföras blockvis, där varje block av data är lika stort som den nyckel som används.

Kryptering av en datamängd A, med en nyckel, som även kan ses som en initieringsvektor (IV), kan ske på följande sätt:

1. Initieringsvektorn IV förhandlas fram mellan sändare och mottagare och hålls hemlig från eventuella angripare.

2. Datamängden A delas upp i block med samma storlek som initieringsvektorn IV.

3. Varje block i datamängden A XOR:as med initieringsvektorn IV.

4. Den krypterade datamängden skickas till mottagaren.

A är nu krypterad med IV som nyckel. För att dekryptera A appliceras nyckeln IV bara på den krypterade datan. Eftersom XOR har egenskapen att den är reversibel kan

den ursprungliga datan återställas utifrån den krypterade datan och den IV som använts vid krypteringen.

Kryptering och dekryptering av ett datablock sker på följande sätt:

- Originalmeddealnde: M
- Nyckel: IV
- Kryptering av meddelandet: $IV \oplus M = C$
- Dekryptering av meddelandet: $C \oplus IV = M$

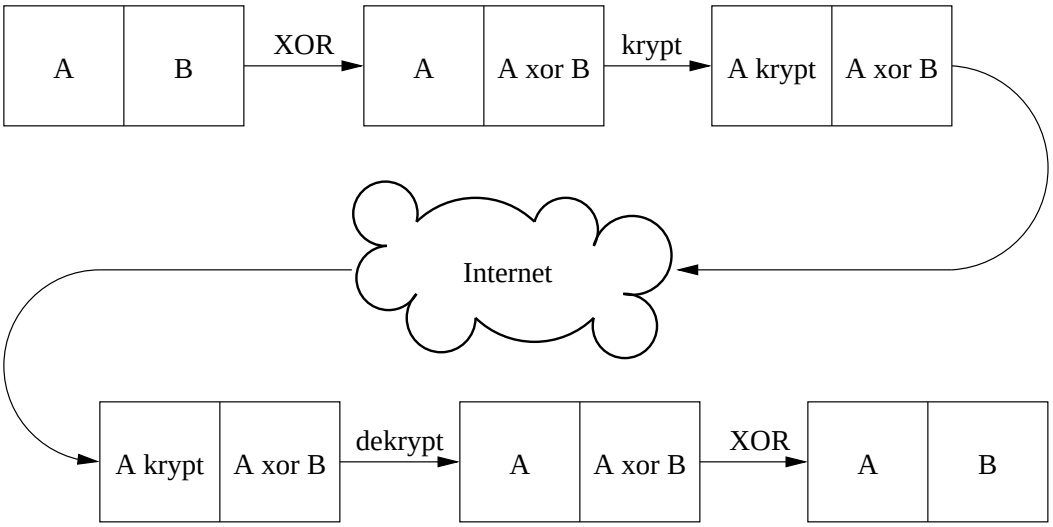
För n stycken bitar i en given nyckel finns det 2^n möjliga värden. Som vi har sett i kapitel 3.3.1 så anses DES med sin nyckel på 56 bitar inte vara säker i dagens läge. Därför rekommenderas en nyckellängd på minst 64 bitar. Detta ger 2^{64} eller 18 446 744 073 709 551 616 olika möjliga nycklar. Val av blocklängd diskuteras mera ingående i kapitel 4.4.3.

XOR-operationen uppfyller således kraven om att vara reversibel, samt att originaldatan förändras så att den inte kan utläsas ur den krypterade datamängden. Vi ska därför fortsättningsvis i detta kapitel ta fram en modell för hur XOR-operationen ska kunna integreras som en algoritm för lättviktskryptering i kombination med partiell säkerhet. Vi kommer sedan i kapitel 5 utföra prestandatester för att se hur en sådan algoritm förhåller sig till konventionella krypteringsalgoritmer.

4.4.2 Modeller för XOR-kryptering

Genom att blockvis använda XOR på den okrypterade datan sker en förändring av datan så att återsynkronicering inte längre är möjlig. Att utföra XOR är även reversibelt, vilket gör att mottagaren kan återställa datan genom att utföra XOR med samma nyckel.

Det värde som används för att utföra XOR-operationen måste dock hållas hemligt för att en angripare inte ska kunna utföra samma operation för att återställa datan. Detta kan ske genom att använda värdet från ett block med krypterad datan innan den krypteras.



Figur 4.7: Kryptering med XOR

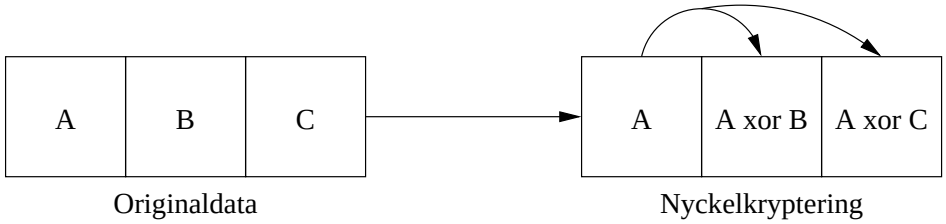
Denna modell kräver att en fil alltid har ett block med starkt krypterad data först i filen. För att öka säkerheten i denna modell kan flera block med starkt krypterad data distribueras över filen. Data från dessa block kan innan kryptering användas som nyckel till XOR-operationen i efterföljande block.

För det enklaste fallet av partiell kryptering kan användaren välja hur stor andel av datamängden som ska krypteras. Den krypterade datan distribueras ut över hela filen. Dessa block kan därmed användas som nyckelblock till lättviktskrypteringen och andelen nyckelblock kan därmed varieras beroende på andelen krypterad data som krypterats med partiell kryptering.

Modellen ger på detta sätt möjlighet att utföra partiell kryptering med en konventionell krypteringsalgoritm, kombinerat med att den erbjuder ett grundskydd för hela datamängden. Figur 4.7 visar hur ett meddelande kan krypteras och skickas via Internet, för att därefter dekrypteras av mottagaren.

Nyckelberoende

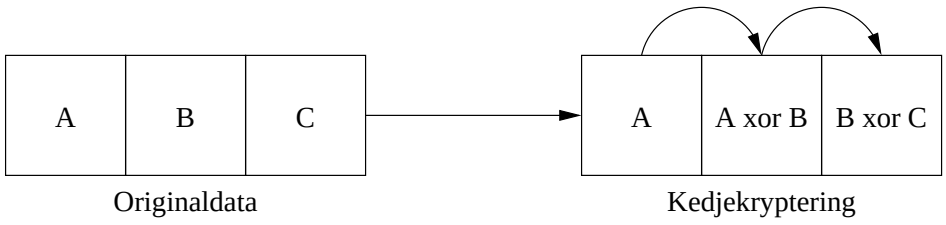
Om datamängden delas upp i ett större antal lika stora block kan det första blocket användas för att utföra XOR-operationen på de övriga blocken. Det första blocket krypteras efter att XOR-operationen utförts på övriga block. På detta sätt blir alla block beroende av det första blocket som har krypterats. Förhållandet mellan blocken i en fil där alla övriga block beror på nyckelblocket visas i figur 4.8.



Figur 4.8: Nyckelberoende kryptering med XOR

Kedjeberoende

En annan variant kan vara att alltid använda det värde som föregående block har innan XOR-operationen utförts på detta. Detta skapar ett kedjeberoende där alla block beror av det föregående. Sambanden mellan blocken vid kedjeberoende illustreras i figur 4.9.



Figur 4.9: Kedjeberoende kryptering med XOR

4.4.3 Säkerhet i lättviktskryptering

Den metod som vi föreslagit för lättviktskryptering har vissa brister i säkerheten gentemot konventionella krypteringsalgoritmer. Om samma nyckel används för kryptering av en fil räcker det med att en angripare känner till innehållet i ett block från originalfilen för att kunna dekryptera hela meddelandet.

Många kända filformat innehåller headerinformation som har samma utseende för samtliga filer av det formatet. Exempelvis kan en JPEG-bild innehålla textsträngarna "JFIF" eller "PHOTOSHOP" i headerinformationen. Om en angripare utför en XOR-operation med en sådan textsträng på rätt ställe i en fil som är krypterad med nyckelberoende så erhålls hela, eller delar av nyckeln för filen i klartext. Denna kan sedan användas för att dekryptera filen. Om krypteringen sker med kedjeberoende kan filen dekrypteras genom att angriparen framåt i filen utför XOR med den matchade textsträngen och bakåt i filen med den erhållna nyckeln.

Attack med känd delmängd av originaldata

Om en fil skyddas med endast lättviktskryptering kan en angripare som endast känner till en delmängd av originalfilen erhålla den initieringsvektor som användes vid krypteringen.

Detta illustreras med följande exempel med originalmeddelande M och initieringsvektor IV.

1. Originalmeddelandet M delas upp i block:

   $M = M_1, M_2, ..., M_{n-1}, M_n$

2. Originaldatan kedjekrypteras med initieringsvektorn:

   $M_c = IV \oplus M_1, M_1 \oplus M_2, ..., M_{n-1} \oplus M_n$

3. Om en angripare känner till ett block M_k i originaldatan kan IV enkelt erhållas genom att utföra XOR på den krypterade datan M_c:

$$IV = (IV \oplus M_1) \oplus (M_1 \oplus M_2) \oplus ... \oplus (M_{k-1} \oplus M_k) \oplus M_k$$

Nyckelblock för ökad säkerhet

För att undvika att hela filens innehåll kan dekrypteras genom att innehållet i ett block är känt kan stark kryptering användas på enstaka block i filen. Genom att sprida ut starkt krypterade block av data förhindras att hela filen kan återskapas om innehållet i ett block är känt. Hur många starkt krypterade block som krävs beror på filens innehåll. Vid kedjekryptering kommer den del av datan som kan dekrypteras bli begränsad till området mellan två starkt krypterade block.

Om det okrypterade innehållet i de starkt krypterade blocken används som nycklar till lättviktskrypteringen innebär det dock att den data som skyddas med stark

kryptering kan återskapas om en angripare knäcker lättviktskrypteringen.

Följande exempel visar hur användandet av så kallade nyckelblock förhindrar att en angripare kan återskapa hela filens innehåll vid en attack när innehållet i ett block är känt. Vi har i detta exempel valt att använda var fjärde block som nyckelblock.

1. Originalmeddelandet delas upp i block:

   $M = M_1, M_2, M_3, M_4, M_5, M_6, M_7, M_8$

2. Meddelandet krypteras med XOR-algoritmen och initieringsvektorn IV:

   $M_x = IV \oplus M_1, M_1 \oplus M_2, M_2 \oplus M_3, M_3 \oplus M_4, M_4 \oplus M_5, M_5 \oplus M_6, M_6 \oplus M_7, M_7 \oplus M_8$

3. Var fjärde block krypteras med en konventionell krypteringsalgoritm:

   $M_{xc} = (IV \oplus M_1)_c, M_1 \oplus M_2, M_2 \oplus M_3, M_3 \oplus M_4, (M_4 \oplus M_5)_c, M_5 \oplus M_6, M_6 \oplus M_7, M_7 \oplus M_8$

Om en angripare får tillgång till det krypterade meddelandet och känner till värdet för ett av blocken i originalfilen kan endast de block som ligger mellan två starkt krypterade block samt det föregående block som skyddades med stark kryptering återställas. Antag att

en angripare känner till innehållet i det tredje blocket M_3 i originalmeddelandet. Följande block ur originalmeddelandet kan då återställas:

- $M_1 = (M_1 \oplus M_2) \oplus (M_2 \oplus M_3) \oplus M_3$
- $M_2 = (M_2 \oplus M_3) \oplus M_3$
- $M_4 = (M_3 \oplus M_4) \oplus M_3$

På grund av att första blocket har krypterats med en konventionell krypteringsalgoritm kan inte IV återskapas med XOR-operationen. På samma sätt kan de block efter det andra starkt krypterade blocket inte heller återställas.

Känslighet för okomprimerade filformat

Problemet med okomprimerade filtyper är mängden information per byte data är lägre. Dessa filer är alltså större än vad de skulle behöva vara ur entropisynvinkel. Detta leder till att statistiska analyser är lättare att utföra eftersom en partiell kryptering kommer att förstöra mindre mängd information. Med andra ord, i okomprimerad data finns ofta mycket redundans som kan utnyttjas eftersom den skapar olika beroenden.

När det gäller multimedia, speciellt strömmande medier är detta sällan ett problem eftersom formatet i sig redan från början är hårt komprimerat för att ge bra kvalitet på låg bitrate. Med bitrate menas hur många bitar som används för att representeras per sekund, vare sig det handlar om film eller musik, högre bitrate ger således bättre kvalitet.

I dessa hårt komprimerade filformat är redundansen låg, vilket innebär att mycket mer information förstörs vid kryptering av ett block. Filformaten är också ofta skapade för att minimera beroenden eftersom detta tar plats och innebär att man för en given filstorlek inte kan ha lika hög bitrate.

Komprimerad data med låg redundans lämpar sig således bra för partiell kryptering då man med små medel kan förstöra mycket. Även om det primära användningsområdet

för partiell säkerhet är just strömmande multimedia var målet att skapa en generell metod som kan fungera även för filformat med högre redundans. För att uppnå detta krävs något som förstör redundansen för den aktuella filen även om dess format är av en sådan typ att redundansen är hög. Som vi nämnde ovan fungerar komprimering mycket bra för att minska redundansen och ta bort beroenden i en fil.

Sammanfattningsvis lämpar sig XOR bättre för filformat som i sig redan är komprimerade, då dessa redan har en låg redundans. När det gäller filformat med hög redundans kan komprimering först utföras, då detta kommer att minska redundansen och erbjuda ett liknande skydd för okomprimerade filformat som för de format som redan är komprimerade.

Blockstorlek

Blaze med flera har skrivit en rapport [BDR+96] om nyckellängder för symmetriska chiffer. För att undvika att en angripare kan knäcka de delar av filen som skyddas med lättviktskryptering bör blocklängden hållas tillräckligt stor för att en attack med "brute force" inte är möjlig att utföra på ett block. Gränsen för detta beror på användarens säkerhetskrav, men en allt för liten blocklängd ger obefintlig säkerhet.

Då lättviktskrypteringen är tänkt att användas tillsammans med en konventionell krypteringsalgoritm är det enklast att anpassa blockstorleken i lättviktskrypteringen till en jämn multipel av den blockstorlek som används i den konventionella krypteringsalgoritmen.

4.5 Riktad kryptering

För kända filformat kan kryptering ske på delar av innehållet. Beroende på vilka delar som krypteras erhålls varierad säkerhet. Säkerheten varierar här beroende på vilka delar som krypteras och kan därför inte delas in på en skala. Här måste avsändaren aktivt besluta vilka delar som ska skyddas. Vi ska i detta kapitel undersöka några av de möjligheter som ges med riktad kryptering. De testar som utfördes på partiell kryptering av data i

kapitel 4.3 kan sägas vara enkla fall av riktad kryptering. I dessa fall tog vi ingen hänsyn till filformatet, utan distribuerade ut krypteringen över all bilddata i filen. För att på ett smartare sätt kunna utföra riktad kryptering krävs att formatet för den krypterade datan är känt. Vi ska därför undersöka JPEG-formatet mera noggrant för att se hur riktad kryptering kan appliceras på JPEG-bilder.

4.5.1 Kryptering av headerdata

Svagheten i den lättviktsalgoritm som presenterades i kapitel 4.4 ligger i första hand i att genom att angriparen känner till vanligt återkommande mönster i filen kan hela filen dekrypteras.

Om en angripare har kännedom om filformatet finns vissa mönster i headerdata som återkommer för alla JPEG-filer. Om bilden har skapats i Photoshop innehåller headern textsträngen "PHOTOSHOP" i klartext. För att undvika att en angripare ska kunna genomföra denna typ av attack så kan headerinformationen krypteras med en konventionell krypteringsalgoritm, därefter kan bilddata krypteras med mindre krävande lättviktskryptering.

För att undvika trafikanalys genom att skicka headerinformation i klartext kan denna i stället skyddas med stark kryptering. Genom att skydda headerdata med stark kryptering kommer informationen om hur bilden komprimerats att skyddas. Detta medför att det huffmanträd som använts för bildkodningen inte kan erhållas

av en angripare. På så sätt undviks att angriparen får tillgång till hur bilddata kodats och statistisk analys av bilddatan försvåras. Detta leder till ökad säkerhet som medför att bilddata kan kodas med en svagare krypteringsalgoritm.

4.5.2 Kryptering av bilddata

Målet med bildkryptering kan i vissa fall vara att enbart förändra bilddata tillräckligt för att bilden inte ska vara användbar för en angripare. Vi ska här visa några tänkbara

varianter för kryptering av bilddata.

Genom att kryptera endast bilddata kan en angripare direkt identifiera vilken trafik som skickas och på så sätt utföra en trafikanalys. Men genom att inte kryptera headerdata undviks möjligheten att knäcka en svag kryptering genom kända mönster i headerinformationen, då denna inte längre har ett beroende mot krypteringen av bilddatan.

Kryptering av DC-koefficienter

För att kunna välja ut specifika delar av bilddatan har vi använt oss av en progressiv JPEG-bild. Denna finns beskriven i appendix A.4.

Anledningen till att vi valt denna bild är att den delar upp bilddata i flera scans. I stället för att bilden läses in block för block så väljs de viktigaste delarna ut och läses in först. Detta gör att bilden gradvis förbättras vid inläsningen.

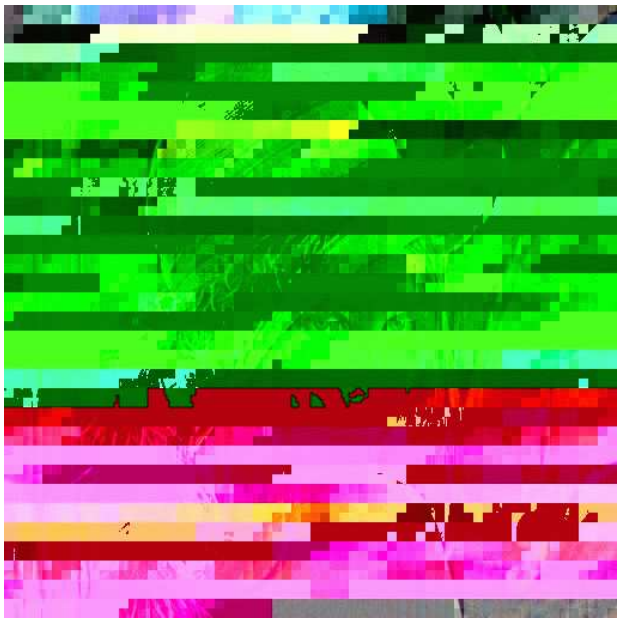
Detta medför att det är möjligt att välja ut de scans som innehåller DC-koefficienterna för bilden. Vi valde således ut de två scans som innehöll koefficienterna noll och krypterade bilddatan för dessa två scans. Resultatet från detta visas i figur 4.10.

Då de krypterade delarna av bilden ökar på det brus som gör att bilden försämras har vi även utfört ett test där DC-koefficienterna togs bort ur filen i stället för att krypteras. Resultatet från detta visas i figur 4.11. Denna bild är lätt att återskapa utifrån en bild med endast krypterade DC-koefficienter då det räcker med att plocka bort den krypterade datan från filen.

4.6 Generell krypteringsmodell

För att kunna utföra partiell kryptering oberoende av filformat krävs att åtminstone okrypterad data modifieras med en algoritm för lättviktskryptering, vilket diskuterades i kapitel 4.4. Hur lättviktskrypteringen utförs kan dock hanteras på två olika sätt.

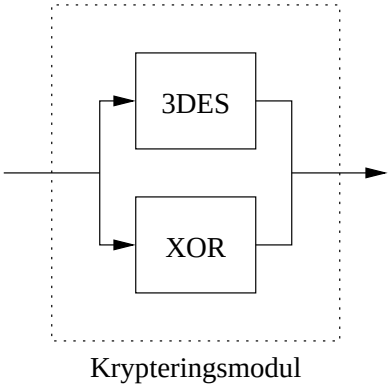
1. Algoritmen för lättviktskrypteringen integreras i den befintliga krypteringsmodellen



Figur 4.10: Progressiv bild med krypterade DC-koefficienter

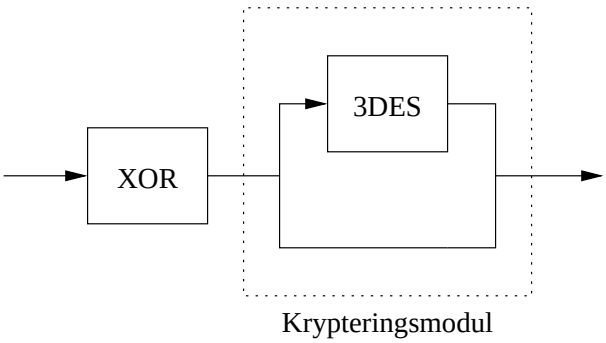
Figur 4.11: Progressiv bild med borttagna DC-koefficienter

och data krypteras antingen med en stark krypteringsalgoritm eller skyddas med lättviktskryptering. Valet av algoritm sker blockvis med ett förutbestämt intervall. Detta illustreras i figur 4.12.



Figur 4.12: Krypteringsmodell ett med integrerad lättviktskryptering

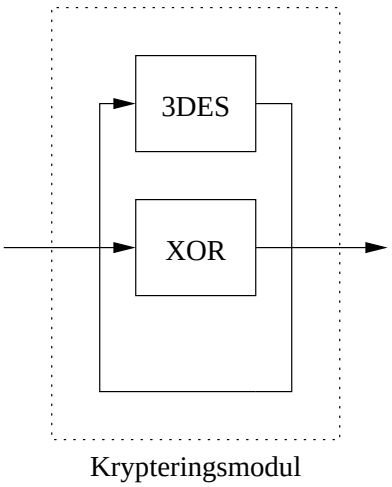
2. Algoritmen för lättviktskryptering ses som en preprocessor som bearbetar hela datamängden innan krypteringen sker på en delmängd av den preprocessade datan. Selektionen innebär i detta fall ett val om data ska krypteras eller inte. Detta illustreras i figur 4.13.



Figur 4.13: Krypteringsmodell två med lättviktskryptering som preprocessor

Om modellen även ska innefatta möjligheten till riktad kryptering krävs ytterligare valmöjligheter, då det kan tänkas att delar av datan i vissa fall inte behöver något skydd.

Detta innebär att ytterligare en valmöjlighet måste införas där data inte processas av någon av algoritmerna. För att inte utföra ett val för om kryptering ska ske i två steg lämpar sig en utökning av modell ett som beskrevs tidigare i detta kapitel. Modellen illustreras i figur 4.14.



Figur 4.14: Generell krypteringsmodell anpassad för riktad kryptering

Förutom att modellen måste utökas med möjligheten att inte modifiera delar av datamängden behöver även möjligheterna att konfigurera hur den krypterade datan ska distribueras över filen utökas då vi visade i kapitel 4.5 att ett fast intervall för vilka block som ska krypteras inte alltid är tillräckligt för att uppnå önskat resultat. För att uppnå detta måste ett mera komplicerat schema för krypteringen specificeras. Ett extremfall av detta vore att specificera individuellt för varje block vilken typ av kryptering som ska användas på blocket.

Kapitel 5

Prestandatester

De tester som vi gjort var att mäta hur mycket data en konventionell krypteringsalgoritm klarar av att hantera per sekund. Vi jämförde sedan dessa tider mot tiden för att kryptera data med en algoritm för lättviktskryptering som beskrivits i kapitel 4.4. Vi är även intresserade av vilka tider som kan erhållas vid partiell kryptering med en konventionell krypteringsalgoritm när denna kombineras med en algoritm för lättviktskryptering.

Genom att mäta tiden för att kryptera en bestämd datamängd går det att beräkna algoritmernas "throughput", det vill säga hur många Megabyte (MB) data de kan bearbeta per sekund på ett givet system. Sådan information är bra att ha om antaganden angående prestandan i andra miljöer önskas. Genom att se på hur algoritmernas throughput förhåller sig till varandra kan antaganden om skillnaden i prestanda göras även på ett annat system.

En algoritm med låg throughput kan innebära begränsningar i systemet. Om en algoritm i ett givet system har en throughput på 5 MB/s ska hantera inkommande data på en datasocket där nätverket i sig klarar av att transportera data i 10 MB/s innebär algoritmens throughput en begränsning i systemet. I detta fall behövs en algoritm som ger en dubbelt så hög throughput för att den inte ska begränsa systemet.

Det finns en del konventionella krypteringsalgoritmer som är intressantare att jämföra med än andra. Dessa är de algoritmer som blivit såpass accepterade att de används i

implementationer av SSL och liknande. Några av dessa krypteringsalgoritmer beskrivs i kapitel 3.3.

5.1 Testplan

För lättviktskryptering måste en egen version implementeras för att kunna genomföra tester på de teorier som presenterats i kapitel 4.4.

Då vår implementation av XOR-algoritmen skrivits i C utan försök till optimeringar i koden ville vi testa detta mot konventionella krypteringsalgoritmer med liknande förutsättningar. Valet av algoritm begränsas därmed något. Då vi i kapitel 3.3 koncentrerade oss på 3DES kändes det naturligt att även utföra prestandatesterna med denna. Valet av implementation föll på Open3DesCrypt [Sch03], då denna var implementerad i C och vi ansåg koden vara relativt enkel att använda för våra syften.

Systemanropet `time` som finns i princip alla *NIX[1] operativsystem användes för tidmätningarna. För att få ett så bra mätvärde som möjligt upprepades varje delförsök 12 gånger, de två yttersta värdena togs bort och medelvärdet av de 10 kvarvarande representerar tiden för det aktuella delförsöket. För att kunna se att 12 mätningar var tillräckligt utfördes även en referenstest med större antal mätningar. Partiell kryptering med 3DES med kryptering av var tionde block valdes för referensmätningarna och utfördes 100 gånger. För att ytterligare minimera felaktiga tider på grund av caching av systemet, sidfel etcetera utfördes testerna på två olika sätt:

1. Det första sättet innebar att delförsöken gjordes utan filsystemet. Indata till algoritmerna togs från `/dev/zero` och den krypterade datan skrevs till `/dev/null`. Dessa två är så kallade virtuella filsystem där hårdvara i form av anrop och söktider hos en hårddisk inte längre sätter en övre gräns på hastighet, utan snarare systemets egna minneshantering. Dessa användes som ett sätt att komma undan påverkan från det

[1] Samlingsnamn för UNIX och Linux operativsystem.

sekundära minnet i testerna och få ett värde på hur snabba algoritmerna är om det var så att filsystemet satte en övre gräns på hastigheten.

2. Det andra sättet var till för att få mätvärden som indikerade prestandan hos algoritmerna vid lokal användning där läs- och skrivtiderna hos en hårddisk spelade in. Filsystemet användes således för alla läs- och skrivoperationer.

5.1.1 Testfall 1, Nullkryptering

Först och främst behövs ett referensvärde som representerar den tid som krävs för att flytta data från inströmmen till utströmmen utan att det utförs några operationer på datamängden. Detta värde sätts genom att mäta tiden det tar att göra en så kallad Nullkryptering.

Att ta med ett dylikt test är lämpligt av flera orsaker. En av dessa är att man kan få en verifikation på om tiden för att läsa data från inströmmen plus tiden för den rena krypteringen faktiskt blir den totala tiden man får vid kryptering från och till filsystemet eller om det är någon mer okänd faktor som spelar in.

5.1.2 Testfall 2, Kryptering med full XOR

Detta testfall avser full kryptering med XOR, det vill säga att hela datamängden XOR:as. Datamängden XOR:as i block av samma storlek som den nyckel som används. Då vi i senare testfall kommer att utföra tester med denna algoritm i kombination med konventionella krypteringsalgoritmer har vi valt att använda en blocklängd på 64 bitar, vilket är samma blocklängd som används av 3DES.

Som kan läsas i kapitel 4.4.1 används den första byten av en given datamängd som IV för att XOR:a resten av datamängden. För att kunna mäta tiderna för enbart lättviktskryptering utan att behöva använda block av starkt krypterad data som nyckel användes en initieringsvektor som startvärde för algoritmen. Vi valde även att implementera XOR-algoritmen så att det skapades ett kedjeberoende mellan blocken där varje block

XOR:ades med den okrypterade datan från föregående block.

5.1.3 Testfall 3, Kryptering med full 3DES

I detta testfallet kommer hela datamängden att krypteras med en krypteringsalgoritm för 3DES. Detta kommer att ge oss möjligheten att beräkna throughput för den implementation av 3DES vi använt oss av. Detta kan sedan jämföras med de uppmätta tiderna för krypteringen med XOR för att få ett förhållande mellan algoritmernas throughput.

5.1.4 Testfall 4, Partiell kryptering med 3DES

Detta kommer att bli det mest intressanta av alla testfallen. Här kommer kryptering med 3DES att utföras på delar av datamängden i kombination med att hela datamängden krypteras med XOR. Genom att jämföra tiderna för denna typ av kryptering med de tider som erhålls från testfall 3 kan ett brytvärde erhållas för hur stor andel data som kan krypteras med 3DES i kombination med XOR och samtidigt få prestandavinster i förhållande till att utföra fullständig kryptering med enbart 3DES.

5.2 Testmiljö

I denna sektion erbjuds en snabb överblick över den hårdvara och mjukvara som användes vid testerna. Detta för att andra ska kunna replikera testerna i liknande miljö om extern verifikation på värden skulle önskas.

- Operativsystem: `Red Hat Linux release 8.0 (Psyche)`
- Kernel: `2.4.18-26.8.0`
- Processor: `Intel(R) Pentium(R) 4 CPU 1.70GHz`
- Intern minne (RAM): `513464 kB PC2100`

- Hårddisk: `Western Digital 40 GB ST340016A`
- Filsystem: `EXT3 FS 2.4-0.9.18`
- Mjukvara
  - `Open3DESCrypt 1.2.1 modifierad`
  - `GCC 3.2 (Red Hat Linux 8.0 3.2-7)`
  - `glibc-2.2.93`
- Flaggor vid kompilering: `-Wall -pedantic -mcpu=pentium4 -O3 -march=pentium4 -ffast-math`

5.3 Testutförande

För varje testfall skrevs ett litet program alternativt modifierades ett tidigare för att utföra testerna. Koden till dessa program skrevs i C och finns tillgänglig i Appendix B.2. All data skrevs och lästes i block om 8 bytes då detta var blocklängden som användes av 3DES. 3DES algoritmen kördes i ECB-mode, vilket förklarades i kapitel 3.3.4. Genom att använda samma blocklängd för alla tester fanns inte risken att några tester var snabbare på grund av snabbare filaccess. De datamängder som testerna gjordes på genererades med systemanropet `dd` och togs från `/dev/zero`. Följande punkter beskriver hur testerna har utförts:

- Testfall ett till och med tre utfördes på filer av varierande storlek. Detta för att se om initieringstiderna för programmen skulle påverka mätningarna. Filer av storleken från 10 kilobyte upp till 100 megabyte användes för dessa tester. Testfall fyra utfördes enbart med filstorleken 10 megabyte.
- Nullkrypteringen gjordes med ett enkelt program som läste från en infil i block om 8 bytes. I de fall som `/dev/zero` och `/dev/null` användes som filsystem riktades

dessa filpekare helt enkelt om. För varje filstorlek kördes detta program 12 gånger och tiderna dokumenterades.

- XOR krypteringen skedde på ett liknande sätt som nullkrypteringen fast datamängden XOR:ades innan den skrevs ner till filsystemet. Första datablocket använde en IV för XOR:andet, följande block använde föregående blocks okrypterade värden som IV.

- För testfall 3 modifierades originalimplementationen på 3DES en hel del då den hade beräkning av checksummor för den krypterade datamängden samt använde CBC-mode för krypteringen. 3DES kräver en initiering för att kunna fungera. Under denna initiering skapas nyckeln som ska användas till krypteringen av hashfunktioner med en krypteringsflagga samt ett lösenord från användaren som indata.

- Den partiella krypteringen av datamängden var en sammanslagning av de två föregående testfallen. Efter att ett givet datablock lästs in XOR:ades det. En inparameter till programmet bestämde hur många block som skulle krypteras med 3DES. Var denna inparameter till exempel 10 så krypterades var tionde datablock innan det skrevs ner till en fildeskriptor.

Efter att de 4 testerna genomförts märktes något underligt med de framtagna tiderna för testfall 4. Tiden för XOR plus full 3DES kryptering var något mindre än tiden för endast full 3DES kryptering. Mätvärdena indikerade således att 3DES kryptering var snabbare om filen XOR:ades först. Eftersom detta var väldigt oväntat så analyserades miljön för att se om det var den som inverkade och var orsaken.

Testfall 3 och 4 gjordes om på två andra datorer som skiljde sig i både hård- och mjukvara. Av dessa tester framgick att det var optimeringsflaggorna[2] vid kompileringen som var orsaken till de konstiga mätvärdena. Även om skillnaden tidsmässigt mellan att slå dessa flaggor av och på var väldigt liten, cirka 0.1 - 0.2 sekunder på 10 MB, så gjorde

[2]Flaggorna som orsakade oväntade värden var -mcpu=pentium4, -march=pentium4 och -ffast-math.

flaggorna att de två testerna bytte plats tidsmässigt. Med flaggorna var således 3DES plus XOR 0.1 till 0.2 sekunder snabbare än ren 3DES, och utan flaggorna var situationen omvänd.

En möjlig orsak till att optimeringsflaggorna gav missvisande värden kan vara att indata tagits från `/dev/zero` vilket innebar att datan bestod av enbart nollor. Det kan ha varit så att den ökade optimeringen märkt detta och därför kunde optimera exekveringen ytterligare. Detta är dock bara spekulationer och något som vi inte lade ner mer tid på. Istället valdes att stänga av flaggorna.

Valet stod då mellan att låta testvärdena från testfall 1 till 3 vara och göra om bara testfall 4 utan dessa flaggor, alternativt att göra om alla testerna. Även om de fel som uppstod i och med användandet av flaggorna inte var speciellt stora så gjordes valet att göra om samtliga tester. För att undvika problemet med flaggorna gjordes testerna utan några som helst optimeringsflaggor. De nya testerna utfördes dock inte i samma omfattning som de första testerna. Endast en filstorlek användes under dessa nya tester. Orsaken till detta var att den valda storleken, 10 MB, var stor nog för att ge bra mätvärden samt att förhållandet mellan olika filstorlekar fortfarande framgick av de första testerna. De nya testerna begränsades också till tester endast mot filsystemet. Alla uppmätta mätvärden, både från de första testerna såväl som de nya finns samlade i Appendix C.

5.4 Förväntade resultat

XOR-operationen används av de flesta krypteringsalgoritmer på marknaden, så även 3DES. Så förutsatt att följande premisser gäller kommer en algoritm som utför endast en XOR-operation på en given datamängd vara snabbare än krypteringsalgoritmerna om:

- Optimeringsnivån på implementationen av den givna krypteringsalgoritmen är lika med eller mindre än den för XOR-funktionen.

- Den givna krypteringsalgoritmen XOR:ar hela den givna datamängden utöver de algoritmspecifika operationerna.

Då vi inte vet om XOR-operationen utförs på hela datamängden i den algoritm vi använder kan vi endast göra ett antagande om att vår implementation av XOR-kryptering borde vara snabbare än krypteringsalgoritmerna.

För att få fram ett teoretiskt värde för hur XOR-algoritmen borde förhålla sig till 3DES krävs att antalet processorcykler som används i denna beräknas. Detta är inget vi lagt ner någon tid på utan prestandatesterna får i stället ge en indikation på hur detta förhållande ser ut.

Förhoppningsvis kommer vår implementation av XOR att vara snabbare än 3DES. Detta ger i så fall att vi kan finna en brytpunkt för hur stor andel data som kan krypteras men 3DES i kombination med att hela datamängden XOR:as.

Kapitel 6

Testresultat

I detta kapitel presenteras resultaten av testerna tillsammans med kommentarer om vilka delar som är mer intressanta än andra. Då många testvärden på flera olika testfall tagits fram kommer inte alla att presenteras, utan snarare ett urval. Uppmätta tider från alla tester finns listade i Appendix C.

6.1 Presentation av resultaten

Skillnaden på att använda ett virtuellt filsystem eller det vanliga visade sig inte spela någon större roll rent tidsmässigt, i flera fall var denna skillnad knappt märkbar. Vi valde därför att använda oss av de uppmätta värden vi erhållit från de tester där filsystemet användes.

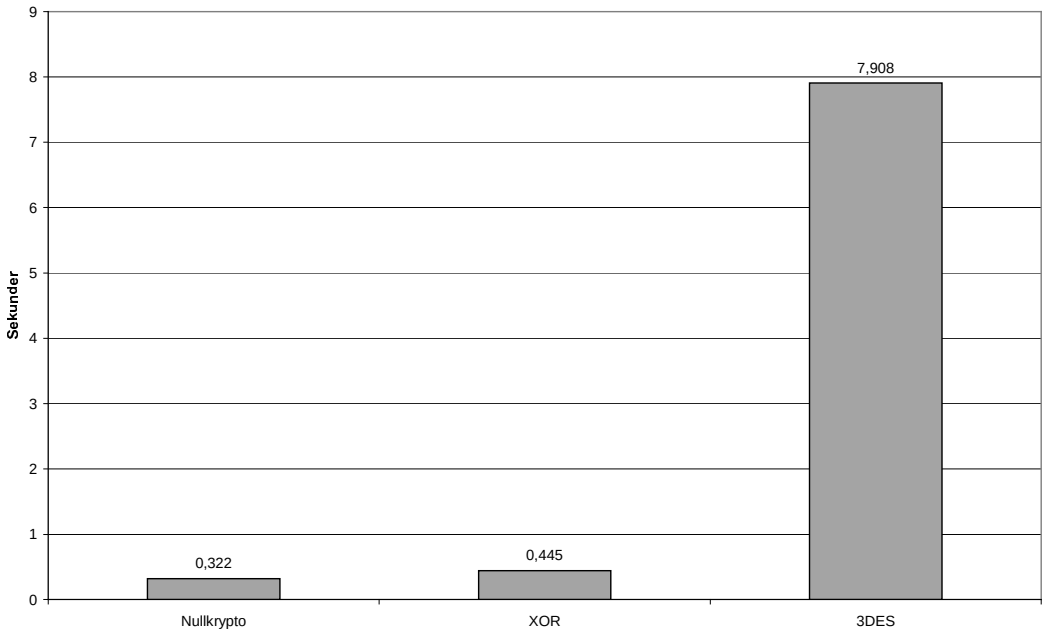
Då vi i kapitel 5.3 beskrev ett problem med kompileringsflaggor, vilket gjorde att några av försöken upprepades kommer vi att använda oss av de nya värdena för presentation av resultaten. Dessa nya försök utfördes endast på filsystemet.

För testfall ett till och med tre uppmättes tider med varierad filstorlek i de ursprungliga försöken. Vid små filstorlekar visade sig tiderna vara näst intill omätbara med den metod vi använde oss av. Eftersom tiderna vid små filstorlekar endast gav tider på några få millisekunder kunde vi göra antagandet att initieringen för programmen inte var en störande

faktor.

6.1.1 Algoritmernas throughput

Testfall ett till tre utfördes för att mäta tiderna för nullkryptering, XOR och 3DES. Detta gjordes för att erbjuda en snabb överblick av de två krypteringsalgoritmerna XOR och 3DES samt sätta dom i perspektiv med nullkryptering. Som beskrevs i kapitel 5.1.1 är nullkryptering tiden för att kopiera data från en filpekare till en annan utan någon form av övriga operationer.



Figur 6.1: Tidsjämförelse mellan nullkryptering, XOR och 3DES.

I figur 6.1 visas jämförelsen mellan dessa tre operationer. Det visar sig direkt att 3DES är flerdubbelt långsammare än de andra två.

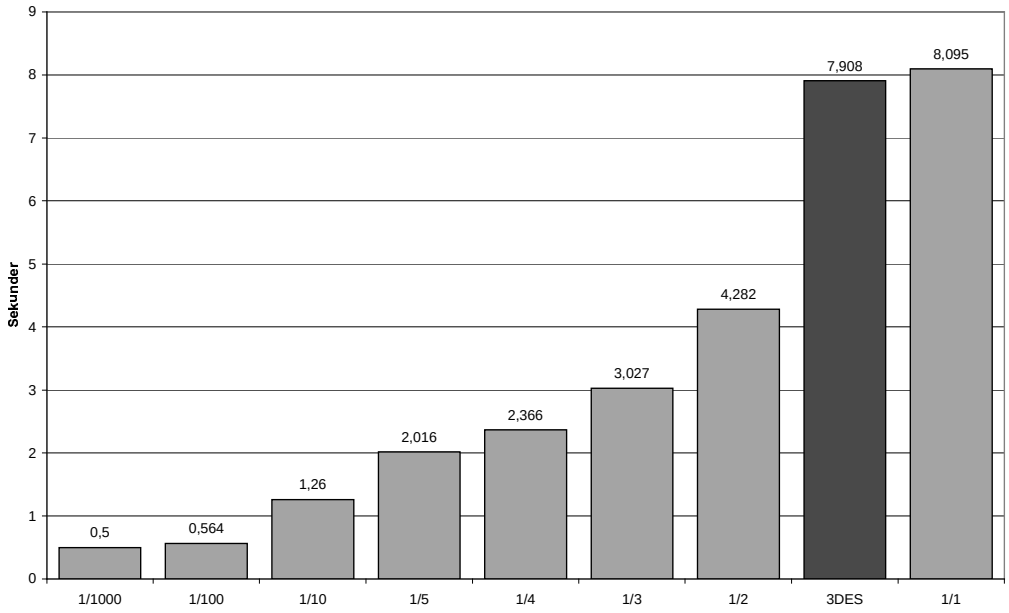
Med hjälp av dessa mätvärden kan throughput beräknas för XOR och 3DES. Tiden för nullkrypto ger den tiden för läs och skrivoperationerna. Då detta utförs även i programmen för XOR och 3DES kan denna tid räknas bort från de övriga för att erhålla den faktiska

tiden för algoritmen. Om dessa beräkningar utförs visar det sig att tiden för 3DES blir 7.586 sekunder och tiden för XOR 0.122 sekunder. Utifrån dessa värden kan algoritmernas throughput beräknas på följande sätt:

- 3DES: $\frac{10}{7.586} \approx 1.3$ MB/s.
- XOR: $\frac{10}{0.122} \approx 82.0$ MB/s.

6.1.2 Partiell kryptering med 3DES

Testfall fyra var att mäta tiderna för partiell kryptering med 3DES av en fil, där hela filen samtidigt krypteras med XOR. De uppmätta tiderna för den partiella krypteringen kan användas till att jämföra med de tester som utfördes i testfall tre, där hela filen krypterades med XOR. Figur 6.2 visar de värden som erhållits från den partiella krypteringen, tillsammans med fullständig kryptering med 3DES från testfall tre.



Figur 6.2: Partiell kryptering med 3DES ovanpå XOR:ad data.

Beteckningarna 1/1000, 1/100, ..., 1/1 anger hur många block utav datamängden som utöver XOR är krypterade med den angivna krypteringsalgoritmen. Således innebär 1/5 i figur 6.2 att datamängden blivit fullt XOR:ad, samt var femte block om 8 byte även krypterats med 3DES. Den stapel med beteckningen 3DES är tiden för endast 3DES från testfall tre.

Utifrån figuren syns att samtliga försök med partiell kryptering var snabbare än att fullständigt kryptera en fil med XOR. Det enda uppmätta värdet med sämre tider var det test när hela filen krypterades med 3DES i kombination med fullständig XOR.

6.1.3 Vad innebär resultaten

Till att börja med har vi bekräftat att den algoritm vi implementerade för lättviktskryptering med XOR är snabbare än båda de krypteringsalgoritmer den jämfördes mot. Detta innebär att den uppfyller kravet om snabbhet som vi ställde på en algoritm för lättviktskryptering i kapitel 4.4. Vi har även visat att partiell kryptering i kombination med denna algoritm kan leda till prestandavinster jämfört mot att endast kryptera data med en konventionell krypteringsalgoritm.

6.1.4 Hur pass säkra är resultaten

Det finns en del faktorer som kan påverka resultaten, eller säkerheten på deras genuitet. Deras chans till påverkan varierar beroende på filstorlek. Dels är det verktyget som användes för att mäta tiden, systemanropet `time` samt antal tester och belastning på systemet när testerna gjordes.

- `time`
- Antal tester
- Belastning på systemet vid testtillfället

När storleken på filerna började sjunka under 1 MB blev tiderna så små att precisionen på tidtagningen inte var tillräcklig. `Time` mäter tiden i millisekunder, och för de tester som utfördes på mindre filer och gav tider på under 5 millisekunder är risken stor att det kan vara svårt att få bra eller pålitliga mättider. Detta problem försvinner dock när storleken på testfilerna höjs så att mättiderna ligger på tiondelar av sekunder och uppåt. Det är mycket på grund av detta som storleken på testfilerna varierades såpass mycket i första testomgången, samt att en storlek på 10 MB valdes för den andra.

Antalet tester som gjordes har självklart en inverkan på hur mycket tillit som bör ges till resultaten. För testerna valdes som nämnts tidigare i kapitel 5.1 att varje enskild test kördes 12 gånger, de två yttervärdena togs bort, för att sedan ta ett medelvärde av de 10 kvarvarande mätningarna. Flera tester skulle säkert gjort att det medelvärde som beräknades skulle kunna ha en större säkerhet att ligga så nära det verkliga värdet som möjligt, men samtidigt var variationerna i testvärdena såpass små att det inte bedömdes som nödvändigt.

För att kunna ha total kontroll över systemets belastning vid testerna hade det varit nödvändigt att starta om testmaskinen i så kallat 'single user mode' samt att stänga av alla icke nödvändiga tjänster i systemet. För att kunna genomföra detta hade det krävts modifikationer på systemets konfiguration samt administratörsrättigheter på det aktuella systemet för att få göra dessa modifikationer. Vid testerna användes en dator som i vanliga fall används av studenter vid universitetet så modifikationer på systemet samt administratörsrättigheter var inte aktuellt. Den aktuella datorn togs dock offline från studentnätet för att minimera resursförbrukning av att andra studenter loggar in på datorn och startar processer. Detta tillsammans med att de två yttervärdena togs bort bedömdes ge en acceptabel precision på testerna.

Referensmätning

Även om de värden som mätts fram verkade stabila fanns fortfarande möjligheten att någonting kan ha påverkat dem, och 12 mätvärden per test är inte tillräckligt för att kunna ge en bra bild av variationsbredden. För att komma åt eventuella avvikelser gjordes därför en referensmätning på en av testerna med en population på 100 istället för 12. Det aritmetiska medelvärdet av detta test förväntades att ligga nära de resterande testerna.

Partiell kryptering med 3DES 1/10 testades således på samma dator 100 gånger. Tiderna dokumenterades och medelvärdet beräknades. Detta aritmetiska medelvärde låg på 1,24s vilket kan jämföras med det medelvärde som kom fram vid bara 12 tester på 1,26s. En skillnad på under 2% innebar en låg spridning så 12 tester ansågs vara tillräkligt för att ge en bra bild av de riktiga tiderna.

Sammanfattningsvis kan sägas att då variationsbredden på de testvärden som framkommit under testerna i nästan alla fall legat under 2% så uppskattas säkerheten på de beräknade medelvärdena som hög.

6.2 Vad gick som väntat

Som väntat var vår implementation av XOR-algoritmen snabbare än den krypteringsalgoritm för 3DES som vi testade mot.

6.2.1 Analys och optimering av XOR

Då testerna har visat att stora prestandavinster kan göras med optimerade algoritmer har vi undersökt vår implementation av XOR på en lägre nivå för att försöka analysera vilka optimeringar som kan vara möjliga. I de tester som gjorts i denna uppsats användes koden som finns i Appendix B.2. Från denna kod finns följande loop:

```
for (i = 0; i < 8; i++) {          (1)
```

```
    tmp[i] = buffer[i] ^ iv[i];    (2)
    iv[i] = buffer[i];             (3)
}                                  (4)
```

Det är denna loop som utför XOR, och således denna loop vi uppmätt tiden för XOR på. Vi ska nu försöka ta reda på hur många processorcykler denna XOR egentligen tar.

Först så måste tre pekare läggas i registren, detta för att hålla reda på var `tmp`, `buffer` och `iv` ligger i minnet. Detta kan göras innan loopen, vilket gör att dess kostnad är noll i sammanhanget. Efter varje loop måste dessa pekare uppdateras, något som görs med `inc`, vilket inte behöver kräva extra klockcykler.

Nu är det dags att beräkna cyklerna som används för varje XOR:ad byte. För varje varv i loopen utförs följande operationer:

- Först så hämtas två byte, `buffer[i]` och `iv[i]`. Detta kostar 2 cykler. (rad 2)
- De två byte ska sedan XOR:as med varandra vilket kostar 1 cykel. (rad 2)
- Denna XOR:ade byte ska sparas undan i `tmp`, 1 cykel. (rad 2)
- Byten i buffer ska sen sparas i `iv`, 1 cykel. (rad 3)

Detta innebär en kostnad på 5 processorcykler per XOR:ad byte data. Alltså 40 cykler för att XOR:a alla 8 byte, plus den extra tid som krävs för att utföra loopen. För att se vad som sker i det program som vi skrivit behöver vi undersöka assemblerkoden för loopen. Om assemblerkoden sparas undan vid kompileringen av källkoden erhålls följande kod för just denna loop:

```
.L144:
        movb (%edx,%ecx), %al
        xorb (%edx,%ebx), %al
        movb %al, (%edx,%edi)
```

```
movb (%edx,%ebx), %al
movb %al, (%edx,%ecx)
incl %edx
cmpl $7, %edx
jle .L144
```

Här kan man räkna antal cykler som krävs, vilket resulterar i 8 cykler per XOR:ad byte plus de extracykler som krävs för hoppet till label `L144`. Koden för XOR har således kompilerats och körs på bytenivå. Detta är dock långt från optimalt. Istället för byte kan alla operationer utföras på word, vilket innebär att loopen i sig är onödig eftersom man kan läsa in 4 byte i taget och öka på offset för det andra varvet. De processorer som producerats av Intel har sedan införandet av MMX (runt Pentium 166/200) klarat av att utföra operationer på 64 bitar, vilket leder till att koden kan optimeras ytterligare.

Slutligen kan alltså sägas att beroende på om 8, 32 eller 64 bitars operationer utförs på datan kommer antal cykler för XOR:andet av 8 byte vara 40, 10 respektive 5. Det finns alltså mycket optimering som kan göras för denna enkla operation.

Kapitel 7

Slutsats

Tack vare de relativt små förändringar som krävs i befintliga system anser vi att partiell säkerhet kan vara ett enkelt sätt för att uppnå dynamiskt konfigurerbar säkerhet med avseende på sekretess och integritet. Då autentisering sker i handskakningsfasen anser vi inte behovet för att på ett dynamiskt sätt kunna kontrollera detta vara lika stort, eftersom autentiseringen inte påverkar prestandan under resten av sessionen. Då målet med uppsatsen var att finna en möjlighet till dynamiskt konfigurerbar säkerhet anser vi att den modell vi presenterat för partiell säkerhet motsvarar målet för de säkerhetsattribut där det är relevant.

7.1 Sammanfattning

För att uppnå dynamisk säkerhet krävs möjligheten för användaren att under en session kontrollera de säkerhetsattribut som påverkar prestandan för applikationen. Av de attribut som beskrivits för hur säkerheten hanteras är det endast sekretess och integritetskontroll som påverkar prestandan för en applikation när en session väl är påbörjad. Sekretess uppnås genom kryptering av den data som skickas och integritetskontroll sker genom beräkning av checksummor. De algoritmer som används för detta kan i många fall vara prestandakrä-

vande.

Av de protokoll som beskrivits har det visat sig att SSL/TLS lämpar sig bättre för dynamiskt konfigurerbar säkerhet än vad IPSec gör. Detta på grund av att IPSec arbetar på en lägre nivå i protokollstacken, vilket gör att säkerheten hanteras transparent för användaren. Säkerheten i IPSec hanteras på ett mera statiskt sätt med konfigurationsfiler, medan SSL/TLS har bättre möjligheter att förhandla om önskad säkerhet mellan sändare och mottagare. Trots att SSL/TLS har möjligheten att förhandla om säkerhetsattributen leder en omförhandling av algoritmer under en pågående session till ett avbrott. Initiering av nya krypteringsalgoritmer kan vara en tidsödande process om detta sker mitt i en pågående session.

Trots de möjligheter som erbjuds till förhandling av säkerhetsattributen i SSL/TLS anser vi inte detta vara tillräckligt för vad som i många fall krävs. De algoritmer som används anses samtliga erbjuda en hög grad av säkerhet och erbjuder ett betydligt bättre skydd än vad som kan vara nödvändigt. Multimediainnehåll med höga prestandakrav har ofta ett lågt ekonomiskt värde och behöver därför inte det starka skydd som algoritmerna medför. Genom att sänka kraven på säkerheten till en lägre nivå kan önskade prestandavinster göras. Detta har lett oss vidare till det område vi benämner som partiell säkerhet.

Partiell säkerhet innebär att endast en delmängd av datan skyddas med en konventionell krypteringsalgoritm, eller att integritetskontroll utförs på endast en delmängd av datan. Idén med partiell säkerhet har testats genom experiment på bilder i TIFF- och JPEG-format. Detta skedde genom att byta ut delar av datamängden mot slumpmässigt vald data för att på så sätt illustrera effekten som erhålls vid partiell kryptering. Vi valde att koncentrera oss på bildkryptering då detta gav möjligheten att visuellt framhäva effekten av den partiella krypteringen.

Då effekten av partiell kryptering är databeroende undersöktes även möjligheten att på ett generellt sätt använda partiell kryptering för att uppnå säkerhet oberoende av filformat. För att skapa dataoberoende introducerades i kapitel 4.4 en ny idé som vi har valt att kalla

lättviktskryptering. Detta skapade möjligheten att behandla data på ett generellt sätt vid partiell kryptering genom att på ett enkelt sätt modifiera datan. Utifrån detta skapades en generell modell för partiell kryptering där lättviktskryptering integrerades för att uppnå dataoberoende. Då prestanda för de algoritmer som används för lättviktskrypteringen är en stor del i hur effektiv den generella modellen för säkerhet är, så har tester utförts på de algoritmer vi tagit fram. Prestandan på dessa har jämförts mot 3DES med mycket gott resultat.

7.2 Framtida arbete

Om en implementation av partiell kryptering ska vara möjlig i ett befintligt

protokoll som till exempel SSL som beskrevs i kapitel 3.1.1 så krävs att en algoritm för lättviktskryptering implementeras. Även handskakningsprocessen bör ses över så att den anpassas till möjligheten att välja krypteringsgrad.

Ett annat angreppssätt för att undvika att förhandla om krypteringsgrad vid handskakningen kan vara att i dataströmmen skicka med extra information om vilka block som ska krypteras. Denna idé medför en ökning av datamängden som skickas, men kan om den används på rätt sätt ge ökade möjligheter att dynamiskt förändra krypteringsgrad under sessionen utan att omförhandlingar behöver ske. Detta har inte närmare undersökts utan lämnas till framtida arbete.

Med den generella krypteringsmodell som presenterades i kapitel 4.6 kan krypteringsscheman anpassas för den data som skickas. Då vi i uppsatsen endast visat på några av de möjligheter som finns för JPEG-bilder så finns det fortfarande mycket arbete kvar i undersökandet av övriga filformat. Även för JPEG-formatet kan fortsatta studier leda till fler och bättre optimeringsmöjligheter för detta format.

Referenser

[Aga01] Christopher D. Agar. Dynamic parameterization of IPSec. Master's thesis, Naval Postgraduate School, Monterey, CA 93943-5000, December 2001.

[Atk95] R. Atkinson. RFC 1825: Security architecture for the internet protocol, August 1995. Status: Proposed Standard.

[BDR+96] M. Blaze, W. Diffie, R. Rivest, B. Schneier, T. Shimomura, E. Thompson, and M. Weiner. Minimal key lengths for symmetric ciphers to provide adequate commercial security. Technical report, Counterpane Systems, 101 East Minnehaha Parkway, Minneapolis, MN 55419, January 1996.

[BR96] R. Baldwin and R. Rivest. RFC 2040: The RC5, RC5-CBC, RC5-CBC-Pad, and RC5-CTS algorithms, October 1996.

[DA99] T. Dierks and C. Allen. RFC 2246: The TLS protocol version 1, January 1999. Status: Proposed Standard.

[DB02] Marc Van Droogenbroeck and Raphaël Benedett. Techniques for a selective encryption of uncompressed and compressed images. In *Proceedings of AC-IVS(Advanced Concepts for Intelligent Vision Systems)*, Ghent, Belgium, September 9-11 2002.

[Ele98] Electronic Frontier Foundation. *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design.* O'Reilly & Associates, Inc., 103a Morris Street, Sebastopol, CA 95472, USA, Tel: +1 707 829 0515, and 90 Sherman Street, Cambridge, MA 02140, USA, Tel: +1 617 354 5800, July 1998.

[Fei73] Horst Feistel. Cryptography and computer privacy. *Scientific American*, 228(5):15–23, May 1973.

[Gar03] Johan Garcia. Jpgparse. `http://www.cs.kau.se/~johang/research/software/jpegparse.html`, 2003. Accessed: 2003-03-22.

[Hal00] Fred Halsall. *Multimedia Communications: Applications, Networks, Protocols and Standards.* Addison-Wesley Publishing, September 2000.

[Hou93] R. Housley. RFC 1457: Security label framework for the internet, May 1993.

[IL00] Cynthia E. Irvine and Timothy E. Levin. Quality of security service. In *Proceedings of the 2000 New Security Paradigms Workshop*, pages 91–99, Ballycotton, County Cork, Ireland, September 19-21, 2000.

[KH98] Thomas Kunkelmann and Uwe Horn. Video encryption based on data partitioning and scalable coding — A comparison. *Lecture Notes in Computer Science*, 1483:95–106, 1998.

[LJ02] Stefan Lindskog and Erland Jonsson. Adding security to quality of service architectures. In *Proceedings of the SSGRR 2002s conference*, L'Aquila, Italy, July 29-August 4, 2002.

[LM91] Xuejia Lai and James L. Massey. Markov ciphers and differential cryptanalysis. In *EUROCRYPT'91*, pages 17–38. Springer-Verlag, 1991.

[PM93] William B. Pennebaker and Joan L. Mitchell. *JPEG Still Image Data Compression Standard*. Van Nostrand Reinhold, New York, NY, USA, 1993.

[Res00] Eric Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison Wesley Professional; 1st edition, October 13 2000.

[Riv94] Ronald L. Rivest. The RC5 encryption algorithm. In Bart Preneel, editor, *Fast Software Encryption: Second International Workshop*, volume 1008 of *Lecture Notes in Computer Science*, pages 86–96, Leuven, Belgium, December 14–16 1994. Springer-Verlag.

[Sch94] B. Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In *Fast Software Encryption, Cambridge Security Workshop Proceedings*, pages 191–204. Springer-Verlag, 1994.

[Sch03] Franz Scheerer. Open3DESCrypt. `http://freshmeat.net/projects/open3descrypt/`, 2003. Accessed: 2003-04-14.

[Sec03] Counterpane Internet Security. Blowfish – Block Cipher Speed Comparisons. `http://www.counterpane.com/speed.html`, 2003. Accessed: 2003-05-16.

[Sta02] William Stallings. *Network Security Essentials – Applications and Standards (Second Edition)*. Prentice Hall, 2002.

[Tea03] OpenSSL Core Team. OpenSSL: The open source toolkit for SSL/TLS. `http://www.openssl.org`, 2003. Accessed: 2003-02-14.

[XN99] Xipeng Xiao and Lionel M. Ni. Internet QoS: A big picture. *IEEE Network Magazine*, pages 28–33, 1999.

[YKS+02] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen. Internet Draft: SSH transport layer protocol, September 20 2002. Status: Internet Draft.

[Zim95] Philip R. Zimmermann. *PGP : Source Code and Internals*. MIT Press, August 1995.

Bilaga A

Bildinformation

Beskrivning av egenskaperna för de bilder som användes för att ge en visuell uppfattning om krypteringsgrad. Informationen från TIFF-bilden har erhållits från programmet tiffinfo. För JPEG-bilderna användes en parser skriven av Johan Garcia [Gar03] som visar liknande information om en JPEG-bild.

JPEG-bilderna skapades genom att originalbilden konverterades till ppm-format med programmet tifftopnm. Därefter användes programmet cjpeg för att konvertera bilderna till JPEG-formatet.

A.1 Originalbild av Lena i TIFF-format

Information om den originalbild vi använde oss av använde oss av visas här genom den information som erhölls från programmet tiffinfo:

```
TIFF Directory at offset 0x8
  Subfile Type: (0 = 0x0)
  Image Width: 512 Image Length: 512
  Resolution: 72, 72 pixels/inch
  Bits/Sample: 8
  Compression Scheme: None
```

```
  Photometric Interpretation: RGB color
  Samples/Pixel: 3
  Rows/Strip: 512
  Planar Configuration: single image plane
```

A.2 Lena i JPEG-format

I detta fall användes inga flaggor till cjpeg. Komprimeringsgraden behölls på standardvärdet 75%.

```
0: SOI (d8)
2: APP0, ( 16 bytes)  JFIF ver 1.01,  X 1 x Y 1 aspect ratio, Thumbnail 0hx0w
20: DQT, ( 67 bytes)  Define 8-bit Quantization Table nr 0:
89: DQT, ( 67 bytes)  Define 8-bit Quantization Table nr 1:
158: SOF0, (17 bytes)  Start of Frame: Baseline
  Precision: 8 bits/sample,Size 512wx512h,3 components
    Component 1: 2x2 HxV sampling, Q-table 0
    Component 2: 1x1 HxV sampling, Q-table 1
    Component 3: 1x1 HxV sampling, Q-table 1
177: DHT, ( 31 bytes)  Define DC huffman table 0
210: DHT, (181 bytes)  Define AC huffman table 0
393: DHT, ( 31 bytes)  Define DC huffman table 1
426: DHT, (181 bytes)  Define AC huffman table 1
609: SOS, ( 12 bytes)  Start of Scan, 3 components in scan
    Component 1: DC Huffman table: 0, AC Huffman table: 0
    Component 2: DC Huffman table: 1, AC Huffman table: 1
    Component 3: DC Huffman table: 1, AC Huffman table: 1
  (Progressive: Coefficients 63-0,  Bits 0-0
37786: EOI,End of Image
Compression Summary: 512hx512v image,3 components at 37788 bytes: 1.153 bits/pixel
 0 Pad (extra FF) and 106 Stuff (0x00 in entropy data) bytes
```

A.3 Lena i JPEG-format med resynkmarkörer

Flaggan -resync 1 användes till cjpeg.

```
0: SOI (d8)
2: APP0, ( 16 bytes)  JFIF ver 1.01,  X 1 x Y 1 aspect ratio, Thumbnail 0hx0w
20: DQT, ( 67 bytes)  Define 8-bit Quantization Table nr 0:
89: DQT, ( 67 bytes)  Define 8-bit Quantization Table nr 1:
158: SOF0, (17 bytes)  Start of Frame: Baseline
  Precision: 8 bits/sample,Size 512wx512h,3 components
    Component 1: 2x2 HxV sampling, Q-table 0
    Component 2: 1x1 HxV sampling, Q-table 1
    Component 3: 1x1 HxV sampling, Q-table 1
177: DHT, ( 31 bytes)  Define DC huffman table 0
210: DHT, (181 bytes)  Define AC huffman table 0
393: DHT, ( 31 bytes)  Define DC huffman table 1
426: DHT, (181 bytes)  Define AC huffman table 1
609: DRI (dd)
615: SOS, ( 12 bytes)  Start of Scan, 3 components in scan
    Component 1: DC Huffman table: 0, AC Huffman table: 0
    Component 2: DC Huffman table: 1, AC Huffman table: 1
    Component 3: DC Huffman table: 1, AC Huffman table: 1
  (Progressive: Coefficients 63-0,  Bits 0-0
1289: RST0 (d0)
1959: RST1 (d1)
2702: RST2 (d2)
3603: RST3 (d3)
4599: RST4 (d4)
5639: RST5 (d5)
6714: RST6 (d6)
7878: RST7 (d7)
8999: RST0 (d0)
10095: RST1 (d1)
11309: RST2 (d2)
```

```
12612: RST3 (d3)
13921: RST4 (d4)
15078: RST5 (d5)
16242: RST6 (d6)
17576: RST7 (d7)
19001: RST0 (d0)
20406: RST1 (d1)
21660: RST2 (d2)
22984: RST3 (d3)
24261: RST4 (d4)
25536: RST5 (d5)
26810: RST6 (d6)
27996: RST7 (d7)
29380: RST0 (d0)
30732: RST1 (d1)
31996: RST2 (d2)
33185: RST3 (d3)
34367: RST4 (d4)
35545: RST5 (d5)
36728: RST6 (d6)
37862: EOI,End of Image
Compression Summary: 512hx512v image,3 components at 37864 bytes: 1.156 bits/pixel
 0 Pad (extra FF) and 95 Stuff (0x00 in entropy data) bytes
```

A.4 Lena i JPEG-format med progressiv scanning

Flaggan -progressive användes till cjpeg.

```
0: SOI (d8)
2: APP0, ( 16 bytes)  JFIF ver 1.01,  X 1 x Y 1 aspect ratio, Thumbnail 0hx0w
20: DQT, ( 67 bytes)  Define 8-bit Quantization Table nr 0:
89: DQT, ( 67 bytes)  Define 8-bit Quantization Table nr 1:
158: SOF2, (17 bytes)  Start of Frame: Progressive
```

```
  Precision: 8 bits/sample,Size 512wx512h,3 components
    Component 1: 2x2 HxV sampling, Q-table 0
    Component 2: 1x1 HxV sampling, Q-table 1
    Component 3: 1x1 HxV sampling, Q-table 1
177: DHT, ( 27 bytes)  Define DC huffman table 0
206: DHT, ( 25 bytes)  Define DC huffman table 1
233: SOS, ( 12 bytes)  Start of Scan, 3 components in scan
    Component 1: DC Huffman table: 0, AC Huffman table: 0
    Component 2: DC Huffman table: 1, AC Huffman table: 0
    Component 3: DC Huffman table: 1, AC Huffman table: 0
  (Progressive: Coefficients 0-0,  Bits 0-1
4137: DHT, ( 44 bytes)  Define AC huffman table 0
4183: SOS, (  8 bytes)  Start of Scan, 1 components in scan
    Component 1: DC Huffman table: 0, AC Huffman table: 0
  (Progressive: Coefficients 5-1,  Bits 0-2
8844: DHT, ( 38 bytes)  Define AC huffman table 1
8884: SOS, (  8 bytes)  Start of Scan, 1 components in scan
    Component 3: DC Huffman table: 0, AC Huffman table: 1
  (Progressive: Coefficients 63-1,  Bits 0-1
9585: DHT, ( 36 bytes)  Define AC huffman table 1
9623: SOS, (  8 bytes)  Start of Scan, 1 components in scan
    Component 2: DC Huffman table: 0, AC Huffman table: 1
  (Progressive: Coefficients 63-1,  Bits 0-1
10349: DHT, ( 52 bytes)  Define AC huffman table 0
10403: SOS, (  8 bytes)  Start of Scan, 1 components in scan
    Component 1: DC Huffman table: 0, AC Huffman table: 0
  (Progressive: Coefficients 63-6,  Bits 0-2
12763: DHT, ( 40 bytes)  Define AC huffman table 0
12805: SOS, (  8 bytes)  Start of Scan, 1 components in scan
    Component 1: DC Huffman table: 0, AC Huffman table: 0
  (Progressive: Coefficients 63-1,  Bits 2-1
19453: SOS, ( 12 bytes)  Start of Scan, 3 components in scan
    Component 1: DC Huffman table: 0, AC Huffman table: 0
```

```
    Component 2: DC Huffman table: 0, AC Huffman table: 0
    Component 3: DC Huffman table: 0, AC Huffman table: 0
  (Progressive: Coefficients 0-0,  Bits 1-0
20238: DHT, ( 34 bytes)  Define AC huffman table 1
20274: SOS, (  8 bytes)  Start of Scan, 1 components in scan
    Component 3: DC Huffman table: 0, AC Huffman table: 1
  (Progressive: Coefficients 63-1,  Bits 1-0
21434: DHT, ( 34 bytes)  Define AC huffman table 1
21470: SOS, (  8 bytes)  Start of Scan, 1 components in scan
    Component 2: DC Huffman table: 0, AC Huffman table: 1
  (Progressive: Coefficients 63-1,  Bits 1-0
22596: DHT, ( 38 bytes)  Define AC huffman table 0
22636: SOS, (  8 bytes)  Start of Scan, 1 components in scan
    Component 1: DC Huffman table: 0, AC Huffman table: 0
  (Progressive: Coefficients 63-1,  Bits 1-0
36403: EOI,End of Image
Compression Summary: 512hx512v image,3 components at 36405 bytes: 1.111 bits/pixel
 0 Pad (extra FF) and 107 Stuff (0x00 in entropy data) bytes
```

Bilaga B

Källkod

B.1 Blockmodifierare

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main (int argc, char **argv)
{
    FILE* ifd, *ufd;
    int start, end, grade, counter=0;
    char buf;
    char a;

    /*Kontrollera att alla argument angivits*/
    if (argc != 6) {
        printf ("Usage: %s startpos endpos grade jpgfile utfile\n", argv[0]);
        return 1;
    }

    ifd = fopen (argv[4], "rb");
```

```
    ufd = fopen (argv[5], "wb");
    start = atoi (argv[1]);
    end = atoi (argv[2]);
    grade = atoi(argv[3] );

    while (fread (&buf, 1, 1, ifd) > 0)
    {
        /*Kontrollera att positionen är ok*/
        if (counter >= start && counter <= end )
        {
            /*Kontrollera om kryptering ska ske
            kryptering sker endast på block som
            är en jämn multipel av grade*/
            if( (grade != 0) && !(counter % grade) )
            {
                /*Byt ut mot ett slumpmässigt värde
                Se till att buf inte kan bli 255
                då detta måste paddas för JPEG*/
                buf = rand() % 255;
        }
        /*Om grade anges till 0 tas blocken
        bort i stället för att krypteras*/
        if( grade != 0 )
            fwrite (&buf, 1, 1, ufd);
        counter++;
    }
    fclose (ifd);
    fclose (ufd);
    return 0;
}
```

B.2 Testkod

B.2.1 Testfall 1, Nullkryptering

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main (int argc, char **argv)
{
  FILE *in_fd;
  FILE *out_fd;
  char buffer[8];
  int len;

  /* Checking inparams */
  if (argc != 3)
    {
      printf ("Usage: %s infile outfile\n", argv[0]);
      exit (1);
    }

  /* Opening files for reading and writing */
  if (!(in_fd = fopen (argv[1], "rb")))
    {
      printf ("Can't open file %s\n", argv[1]);
      exit (1);
    }
  if (!(out_fd = fopen (argv[2], "wb")))
    {
      printf ("Can't open file %s\n", argv[2]);
      fclose (in_fd);
      exit (1);
```

```
    }

  /* Reading from infile to outfile */
  while ((len = fread (buffer, 1, 8, in_fd)))
    {
      while (len < 8)
        buffer[len++] = 0;

      fwrite (buffer, 8, 1, out_fd);
    }

  /* Closing files */
  fclose (in_fd);
  fclose (out_fd);

  return 0;

}
```

B.2.2 Testfall 2, Full kryptering med XOR

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main (int argc, char **argv)
{
  FILE *in_fd;
  FILE *out_fd;
  char buffer[8];
  char tmp[8];
  char iv[8];
  int len, i;
```

```
/* Checking inparams */
if (argc != 4)
  {
    printf ("Usage: %s infile outfile iv\n", argv[0]);
    exit (1);
  }

/* Reading IV for the XOR function */
strncpy (iv, argv[3], 8);

/* Opening files for reading and writing */
if (!(in_fd = fopen (argv[1], "rb")))
  {
    printf ("Can't open file %s\n", argv[1]);
    exit (1);
  }
if (!(out_fd = fopen (argv[2], "wb")))
  {
    printf ("Can't open file %s\n", argv[2]);
    fclose (in_fd);
    exit (1);
  }

/* Reading from infile to outfile */
while ((len = fread (buffer, 1, 8, in_fd)))
  {
    while (len < 8)
      buffer[len++] = 0;

    /* Performing XOR */
    for (i = 0; i < 8; i++)
      {
```

```
          tmp[i] = buffer[i] ^ iv[i];
          iv[i] = buffer[i];
        }

      fwrite (tmp, 8, 1, out_fd);
    }

  /* Closing files */
  fclose (in_fd);
  fclose (out_fd);

  return 0;

}
```

B.2.3 Testfall 3, Full kryptering med 3DES

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <time.h>
#include <ncurses.h>
#include "d3des.h"
#include "sha1.h"

void burnKey (uint8_t * key)
{
  memset (key, 0, 24);
  des3key (key, EN0);
  des3key (key, DE1);
}
```

```
int main (int argc, char **argv)
{
  FILE *in;
  FILE *out;
  char pwd[8];
  char digest[20];
  char k_tot[24];
  char buf[8];
  char tmp[16];
  int len;
  SHA1Context sha;

  /* Checking inparams */
  if (argc != 4)
    {
      printf ("Usage: %s infile outfile password\n", argv[0]);
      exit (1);
    }

  /* Opening files for reading and writing */
  if (!(in = fopen (argv[1], "rb")))
    {
      printf ("Can't open file %s\n", argv[1]);
      exit (1);
    }
  if (!(out = fopen (argv[2], "wb")))
    {
      printf ("Can't open file %s\n", argv[2]);
      fclose (in);
      exit (1);
    }

  /* Reading pass from inparams and preparing it */
```

```
strcpy (pwd, argv[3]);
SHA1Reset (&sha);
SHA1Input (&sha, pwd, strlen (pwd));
SHA1Result (&sha, digest);
memcpy (k_tot, digest, 4);

SHA1Reset (&sha);
SHA1Input (&sha, digest, 20);
SHA1Result (&sha, digest);
memcpy (k_tot + 4, digest, 20);

/* Cleaning up memory used for password creation */
memset (pwd, 0, sizeof (pwd));

/* Creating the key for encryption mode */
des3key (k_tot, EN0);

while ((len = fread (buf, 1, 8, in)))
  {
    while (len < 8)
      buf[len++] = 0;

    Ddes (tmp, tmp);

    fwrite (tmp, 8, 1, out);
  }

/* Closing files */
fclose (in);
fclose (out);

/* Destroy the key used for encryption */
burnKey (k_tot);
```

```
  return 0;

}
```

B.2.4 Testfall 4, Partiell kryptering med 3DES

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <time.h>
#include <ncurses.h>
#include "d3des.h"
#include "sha1.h"

void burnKey (uint8_t * key)
{
  memset (key, 0, 24);
  des3key (key, EN0);
  des3key (key, DE1);
}

int main (int argc, char **argv)
{
  FILE *in;
  FILE *out;
  uint8_t pwd[256] = "";
  uint8_t digest[20];
  uint8_t k_tot[24];
  char iv[] = { "kalle123" };
  char buf[8];
  char tmp[16];
```

```
int len, i;
SHA1Context sha;
long rounds;
int c_level;

/* Checking inparams */
if (argc != 5)
  {
    printf ("Usage: %s infile outfile password cryptlevel\n", argv[0]);
    exit (1);
  }

c_level = atoi (argv[4]);

/* Opening files for reading and writing */
if (!(in = fopen (argv[1], "rb")))
  {
    printf ("Can't open file %s\n", argv[1]);
    exit (1);
  }
if (!(out = fopen (argv[2], "wb")))
  {
    printf ("Can't open file %s\n", argv[2]);
    fclose (in);
    exit (1);
  }

/* Reading pass from inparams and preparing it */
strcpy (pwd, argv[3]);
SHA1Reset (&sha);
SHA1Input (&sha, pwd, strlen (pwd));
SHA1Result (&sha, digest);
memcpy (k_tot, digest, 4);
```

```
SHA1Reset (&sha);
SHA1Input (&sha, digest, 20);
SHA1Result (&sha, digest);
memcpy (k_tot + 4, digest, 20);

/* Cleaning up memory used for pass and digest */
memset (pwd, 0, sizeof (pwd));
memset (digest, 0, sizeof (digest));

/* Creating the key for encryption mode */
des3key (k_tot, EN0);

rounds = 1;

while ((len = fread (buf, 1, 8, in)))
  {

    while (len < 8)
      buf[len++] = 0;

    for (i = 0; i < 8; i++)
      {
        tmp[i] = buf[i] ^ iv[i];
        iv[i] = buf[i];
      }

    if (rounds++ % c_level == 0)
      Ddes (tmp, tmp);

    fwrite (tmp, 8, 1, out);
  }
```

```
  /* Closing files */
  fclose (in);
  fclose (out);

  /* Destroy the key used for encryption */
  burnKey (k_tot);

  return 0;

}
```

Bilaga C

Mätdata

10k	25K	100k	1 000k	10 000k	100 000k
0.004	0.002	0.004	0.035	0.307	3.025
0.000	0.002	0.006	0.031	0.303	3.033
0.002	0.004	0.004	0.031	0.303	3.051
0.002	0.002	0.004	0.031	0.301	3.031
0.006	0.002	0.006	0.033	0.305	3.023
0.004	0.006	0.004	0.029	0.387	3.043

Tabell C.1: Testfall 1, Nullkrypto med variabel filstorlek, från /dev/zero till /dev/null

10k	100k	1 000k	10 000k	100 000k
0.002	0.004	0.029	0.338	3.176
0.002	0.002	0.037	0.297	3.215
0.002	0.008	0.029	0.320	3.164
0.000	0.004	0.039	0.316	3.145
0.002	0.004	0.039	0.309	3.207
0.002	0.002	0.035	0.326	3.152
0.000	0.004	0.035	0.326	3.148
0.002	0.006	0.033	0.301	3.098
0.000	0.004	0.029	0.322	3.113
0.002	0.002	0.027	0.318	3.150
0.002	0.004	0.037	0.303	3.162
0.000	0.004	0.031	0.332	3.191

Tabell C.2: Testfall 1, Nullkrypto med variabel filstorlek, från filsystemet till filsystemet

10k	100k	1 000k	10 000k	100 000k
0.002	0.006	0.033	0.348	3.313
0.002	0.004	0.037	0.336	3.350
0.000	0.004	0.033	0.334	3.303
0.000	0.004	0.037	0.332	3.334
0.002	0.004	0.035	0.340	3.326
0.000	0.008	0.033	0.363	3.377
0.004	0.006	0.039	0.332	3.330
0.002	0.002	0.039	0.336	3.338
0.004	0.004	0.033	0.332	3.365
0.004	0.008	0.035	0.336	3.342
0.002	0.006	0.035	0.326	3.311
0.002	0.006	0.033	0.332	3.332

Tabell C.3: Testfall 2, Full XOR med variabel filstorlek, från /dev/zero till /dev/null

10k	100k	1 000k	10 000k	100 000k
0.002	0.004	0.033	0.352	3.533
0.002	0.002	0.037	0.363	3.518
0.000	0.006	0.033	0.352	3.445
0.000	0.006	0.043	0.369	3.504
0.000	0.004	0.029	0.379	3.508
0.002	0.006	0.045	0.379	3.533
0.000	0.004	0.035	0.354	3.516
0.000	0.006	0.041	0.381	3.396
0.000	0.004	0.035	0.350	3.484
0.002	0.004	0.039	0.385	3.496
0.002	0.002	0.027	0.381	3.484
0.000	0.006	0.043	0.340	3.445

Tabell C.4: Testfall 2, Full XOR med variabel filstorlek, från filsystemet till filsystemet

10k	100k	1 000k	10 000k	100 000k
0.012	0.080	0.799	8.045	79.520
0.012	0.078	0.793	7.951	79.338
0.008	0.082	0.795	7.998	79.568
0.010	0.078	0.799	7.949	79.471
0.010	0.084	0.799	7.955	79.340
0.010	0.082	0.793	7.947	79.498
0.008	0.082	0.807	7.939	79.338
0.008	0.082	0.805	7.953	79.754
0.010	0.082	0.787	7.949	79.480
0.010	0.082	0.803	8.031	79.459
0.010	0.082	0.795	7.955	79.482
0.008	0.086	0.797	7.951	79.469

Tabell C.5: Testfall 3, Full 3DES med variabel filstorlek, från /dev/zero till /dev/null

10k	100k	1 000k	10 000k	100 000k
0.010	0.074	0.797	7.924	79.314
0.008	0.084	0.795	7.904	79.023
0.010	0.078	0.811	7.955	79.697
0.012	0.080	0.795	7.912	79.232
0.010	0.082	0.785	7.871	79.240
0.010	0.080	0.789	7.939	78.957
0.008	0.080	0.795	7.969	79.234
0.010	0.084	0.793	7.949	79.277
0.010	0.080	0.791	7.922	79.195
0.008	0.084	0.797	7.967	78.840
0.010	0.080	0.789	7.922	78.652
0.008	0.080	0.797	7.932	79.199

Tabell C.6: Testfall 3, Full 3DES med variabel filstorlek, från filsystemet till filsystemet

1/1000	1/100	1/10	1/5	1/4	1/3	1/2	1/1
0.396	0.455	1.129	1.920	2.285	2.902	4.131	7.857
0.414	0.451	1.131	1.902	2.275	2.920	4.152	7.834
0.391	0.453	1.137	1.998	2.445	2.916	4.146	7.828
0.400	0.436	1.139	1.918	2.291	2.926	4.262	7.836
0.400	0.432	1.133	1.922	2.260	2.916	4.152	7.832
0.365	0.428	1.141	1.889	2.271	2.906	4.152	7.867
0.457	0.463	1.133	1.900	2.273	2.904	4.160	7.973
0.396	0.438	1.152	1.930	2.291	2.896	4.160	7.859
0.363	0.443	1.139	1.924	2.266	2.895	4.146	7.859
0.398	0.520	1.143	1.900	2.258	2.963	4.145	7.834
0.379	0.521	1.156	1.906	2.295	2.904	4.160	7.857
0.375	0.455	1.137	1.906	2.260	2.908	4.186	7.844

Tabell C.7: Testfall 4, Varierad andel kryptering med 3DES, filstorlek 10MB

Nullkrypto	XOR	3DES
0.320	0.459	7.842
0.322	0.438	7.924
0.332	0.449	8.059
0.313	0.439	8.121
0.314	0.438	7.869
0.330	0.479	7.852
0.334	0.430	7.846
0.324	0.441	7.893
0.320	0.443	8.092
0.307	0.455	7.838
0.324	0.441	7.861
0.314	0.443	7.840

Tabell C.8: Fullständig kryptering med olika algoritmer, filstorlek 10MB

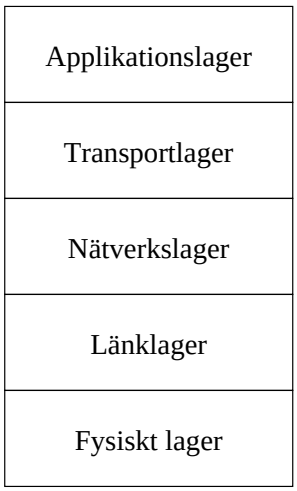
1/1000	1/100	1/10	1/5	1/4	1/3	1/2	1/1
0.488	0.557	1.244	2.008	2.357	3.043	4.289	8.074
0.553	0.545	1.268	2.033	2.348	3.031	4.281	8.074
0.496	0.553	1.275	2.012	2.346	3.098	4.264	8.092
0.508	0.527	1.254	2.000	2.369	3.029	4.264	8.293
0.510	0.574	1.240	2.018	2.404	3.016	4.320	8.100
0.494	0.576	1.262	1.994	2.297	3.016	4.268	8.049
0.496	0.572	1.266	2.002	2.361	3.025	4.316	8.061
0.502	0.580	1.264	2.076	2.346	3.031	4.295	8.043
0.496	0.574	1.254	2.008	2.402	3.023	4.262	8.080
0.492	0.561	1.271	2.012	2.396	3.012	4.293	8.145
0.498	0.564	1.260	2.035	2.396	3.039	4.277	8.193
0.506	0.564	1.254	2.033	2.334	3.008	4.271	8.078

Tabell C.9: Varierad andel kryptering med 3DES utan optimeringsflaggor, filstorlek 10MB

Bilaga D

Internets protokollstack

Internets protokollstack är uppdelad i fem lager. Dessa lager beskrivs i figur D.1. De säkerhetsmekanismer som tas upp i denna uppsats implementeras i de tre översta lagren. I det översta lagret hanteras applikationsprotokoll som till exempel ftp, smtp och http. I nästa lager hanteras transportprotokoll såsom tcp och udp. I tredje lagret hanteras nätverksprotokoll där ip är det vanligast förekommande på Internet.



Figur D.1: Internets protokollstack