Examensarbete

HÖGSKOLAN I SKÖVDE

Krypteringsalgoritmers påverkan på paketstorlek och exekveringstid

Encryption algorithms' impact on packet size and execution time

Examensarbete inom huvudområdet
Informationsteknologi
Grundnivå 15 högskolepoäng
Vårtermin 2025

Alexander Sjögren
William Reutmer

Handledare: Erik Sjöstrand
Examinator: Sanny Syberfeldt

Sammanfattning

Arbetet undersöker hur olika krypteringsalgoritmer påverkar exekveringstid och paketstorlek. Faktorer såsom blockstorlek, krypteringsnyckelns storlek, och antal rundor som algoritmen körs beaktas. Systematiska experiment har utförts i en egenskapad spelprototyp där exekveringstid och paketstorlek mäts. De algoritmer som jämförs är AES, DES, 3DES, Blowfish och Twofish. Två olika implementationer av algoritmerna jämförs, en från .NET och en från Bouncy Castle.

Resultatet visar att AES presterar bäst av alla algoritmer följt av Twofish, sedan Blowfish och 3DES, sist kommer DES som är långsammast av alla algoritmer. Ökning av paketstorlek beror enbart på algoritmernas blockstorlek och påverkas inte av någon annan faktor. Implementationerna från Bouncy Castle presterar överlag bättre än implementationerna från .NET.

Framtida arbete föreslås fokusera på att bredda detta arbete genom att utföra tester med flera algoritmer och i olika spelmotorer. Andra aspekter som skulle kunna undersökas är minnesanvändning eller säkerhet hos algoritmerna.

Nyckelord: Kryptering, Exekveringstid, Paketstorlek, Nätverk, .NET, Bouncy Castle

Innehållsförteckning

1 Introduktion 1

2 Bakgrund 2
2.1 Krypteringsalgoritmer 2
2.2 Tidigare arbeten 4
2.3 Spelprototyp 5

3 Problemformulering 6
3.1 Metodbeskrivning 6
3.2 Metoddiskussion 8

4 Genomförande 11
4.1 Utformning av testmiljö 11
4.2 Resultat 11
4.3 Analys 14
4.4 Slutsats 17

5 Sammanfattning och diskussion 18
5.1 Sammanfattning 18
5.2 Diskussion 18
5.3 Samhälleliga och etiska aspekter 20
5.4 Framtida arbete 21

Referenser 22

1 Introduktion

Krypteringsalgoritmer är viktiga för att skydda känslig information som skickas över nätet. Inom spel kan kryptering användas för att förhindra fusk och det är även särskilt viktigt att algoritmer som används är tidseffektiva, för att inte ta upp den begränsade exekveringstiden som finns allokerad för en bildruta. När data skickas över nätet är det även viktigt att beakta paketstorleken. I detta arbete utförs en jämförelse av de mest populära symmetriska krypteringsalgoritmerna, vilket har identifierats som AES, DES, 3DES, Blowfish och Twofish.

Huvudsyftet med denna studie är att undersöka olika krypteringsalgoritmers påverkan på datamängd, krypterings- och dekrypteringstid när data förbereds för att skickas över ett nätverk. De faktorer som undersöks är exekveringstid och paketstorlek. Olika faktorer som kan påverka undersöks också, såsom storlek på krypteringsnyckel, antal rundor som algoritmerna kör och blockstorlek för de olika algoritmerna. Olika implementationer av algoritmerna har även testats där det är möjligt, för att undersöka möjliga skillnader mellan dem.

De valda krypteringsalgoritmerna, AES, DES, 3DES, Blowfish och Twofish, har implementerats i en egenskapad spelprototyp skapad i Unity (Unity Technologies, 2024a) inspirerad av spelet *RISK: Global Domination* (2020). Implementationer från Cryptography Namespace i C# (Microsoft Learn, 2025b) och paketet Bouncy Castle (2025) har använts. Algoritmernas krypteringstid, dekrypteringstid och storleksökning efter kryptering har mätts genom systematiska tester över 100 olika problemstorlekar och sedan analyserats. Resultatet presenteras främst genom algoritmernas *throughput*. Analysen visar att både nyckelstorlek och antalet rundor påverkar exekveringstiden beroende på algoritm. Överlag bedöms det att AES har högst *throughput* tätt följt av Twofish. 3DES presterar överlag sämst av alla algoritmer. Resultatet tyder även på att implementationerna tillgängliga i Bouncy Castle är bättre optimerade än dess motsvarigheter i .NET.

2 Bakgrund

Kryptering är ett viktigt verktyg när det gäller att hålla information säker. Detta är särskilt viktigt för information som skickas över ett nätverk, där obehöriga individer bör förhindras från att läsa eller modifiera den. I datorspel innebär detta att fuskare kan modifiera eller läsa av information för att få orättvisa fördelar. Enligt en studie utförd av YouGov (Irdeto, 2018) har 60% av onlinespelare påverkats negativt av fuskare. Dessutom uppgav 77% av spelarna att det var sannolikt att de skulle sluta spela ett spel om de trodde att andra spelare fuskade för att få orättvisa fördelar. Fusk kan därmed leda till en förlust av spelare och i sin tur ekonomiska förluster för spelutvecklare. Det är därför viktigt för spelutvecklare att förhindra fusk, och kryptering är ett effektivt verktyg som kan användas i detta syfte.

2.1 Krypteringsalgoritmer

Krypteringsalgoritmer konverterar läsbar information till oläsbar information, även kallat chiffertext. För detta krävs en krypteringsnyckel som också används för att konvertera chiffertexten tillbaka till läsbar information (GeeksforGeeks, 2025b). De algoritmer som valts för detta arbete är blockchiffer, vilket betyder att de körs på block av text av en viss storlek. Om texten är längre än blockstorleken delas den upp i flera block. Om texten är kortare än ett block läggs istället tom data till för att fylla ut till ett helt block. Om algoritmen har en blockstorlek av 128 bitar, kommer den resulterande chiffertextens storlek alltid vara en multipel av 128 bitar. Varje algoritm körs sedan i ett antal rundor, där varje runda ser likadan ut och består av ett antal steg som modifierar texten i ett block. Tiden det tar att utföra kryptering och dekryptering beror på antalet block samt antalet rundor som algoritmen genomför. En annan faktor som påverkar är nyckelstorleken som algoritmerna använder. En större nyckelstorlek är säkrare, eftersom det ökar antalet möjliga nycklar, men kan påverka prestanda negativt (GeeksforGeeks, 2023a).

De krypteringsalgoritmer som undersöks i denna studie är Advanced Encryption Standard (AES), Data Encryption Standard (DES), Triple Data Encryption Standard (3DES), Blowfish och Twofish. De är symmetriska krypteringsalgoritmer, vilket betyder att samma nyckel används för kryptering och dekryptering. En översikt av algoritmerna och deras egenskaper kan ses i Figur 1.

Egenskap	AES	DES	3DES	Blowfish	Twofish
Blockstorlek (bitar)	128	64	64	64	128
Nyckelstorlekar (bitar)	128/192/256	56	192	32-448	128/192/256
Rundor (st)	10/12/14	16	48	16	16

Figur 1: Översikt av valda krypteringsalgoritmer

2.1.1 Advanced Encryption Standard (AES)

AES är en av de populäraste krypteringsalgoritmerna idag. Den utvecklades mellan 1997 och 2001 under en tävling som anordnades av NIST. Syftet med tävlingen var att skapa en krypteringsalgoritm som skulle kunna skydda känslig information långt in på 2000-talet. Av fem finalister valdes Rijndael-algoritmen[1] som vinnare och klassificerades 2001 som en standard av amerikanska myndigheter (National Institute of Standards and Technology (NIST), 2024).

AES kan användas med tre olika nyckelstorlekar, 128, 192 eller 256 bitar. Antalet rundor som körs beror på den valda storleken på nyckeln – för nyckel med storlek 128 bitar kör algoritmen tio rundor, nyckel med 192 bitar kör tolv rundor, och nyckel med 256 bitar kör 14 rundor. Sambandet mellan nyckelstorlek och antalet rundor algoritmen kör kan ses i Figur 2. AES arbetar på block med en storlek av 128 bitar (Panda & Nag, 2015; Ramesh & Suruliandi, 2013).

Nyckelstorlek (bitar)	128	192	256
Rundor (st)	10	12	14

Figur 2: Nyckelstorlek och rundor för AES

2.1.2 Data Encryption Standard (DES)

DES utvecklades på 70-talet av IBM och klassificerades kort därefter som en standard av amerikanska myndigheter. DES var den mest använda krypteringsalgoritmen i över 20 år innan den byttes ut av AES (Coppersmith m. fl., 1997). Idag anses DES inte längre säker då den redan 1998 kunde knäckas på under 3 dagar (Electronic Frontier Foundation (EFF), 1998). Trots detta är algoritmen intressant att undersöka för användningsområden där data endast behöver hållas hemlig under kortare perioder.

DES använder krypteringsnycklar med en initial storlek på 64 bitar, som sedan minskas till 56 bitar genom att slänga var 8:e bit (GeeksforGeeks, 2023b). Algoritmen kör 16 rundor på block med storlek 64 bitar (Ramesh & Suruliandi, 2013; Yassein m. fl., 2017).

2.1.3 Triple Data Encryption Standard (3DES)

3DES föreslogs av Merkle & Hellman (1981) som ett försök att göra DES säkrare, och algoritmen klassificerades officiellt som en standard av amerikanska myndigheter. Idag anses 3DES inte längre som säker, och standarden drogs tillbaka 2024 (National Institute of Standards and Technology (NIST), 2017).

[1] Algoritmen kallades ursprungligen Rijndael, men är idag mer känd som AES.

3DES använder krypteringsnycklar med en sammanlagd storlek på 192 bitar. Algoritmen är en modifierad version av DES, med den skillnaden att den körs tre gånger. Algoritmen appliceras i tre steg: först används algoritmen med en första nyckel för att kryptera texten, sedan används en andra nyckel för att dekryptera texten, och slutligen används en tredje nyckel för att kryptera texten igen. Resultatet är att algoritmen är säkrare, men också tre gånger långsammare än DES. Precis som DES, körs 3DES på block med storlek 64 bitar (Yassein m. fl., 2017).

2.1.4 Blowfish

Blowfish skapades 1993 som ett alternativ till den då föråldrade DES algoritmen. Syftet var att skapa en algoritm som inte var proprietär, påverkad av amerikanska exportrestriktioner eller patenterad, vilket många andra algoritmer då var (Schneier, 1994).

Blowfish kan använda krypteringsnycklar med varierande storlekar från 32 bitar till 448 bitar. Oberoende av krypteringsnyckelns storlek kör algoritmen 16 rundor. Blowfish krypterar block med en storlek på 64 bitar (Panda & Nag, 2015; Ramesh & Suruliandi, 2013).

2.1.5 Twofish

Utvecklingen av Twofish började som ett försök att skapa en förbättrad version av Blowfish (Schneier m. fl., 1998). Algoritmen skapades under AES-tävlingen som anordnades av NIST och blev en av fem finalister, men valdes inte som vinnare (National Institute of Standards and Technology (NIST), 2024).

Twofish kan användas med tre olika nyckelstorlekar, 128, 192, eller 256 bitar. Algoritmen körs i 16 rundor och på block med storlek 128 bitar (GeeksforGeeks, 2024).

2.2 Tidigare arbeten

Krypteringstiden för AES, DES, 3DES och Blowfish har undersökts av Nadeem & Javed (2005) på två olika datorer. Resultatet visar att Blowfish är snabbast av alla algoritmer följt av DES. Likt Nadeem & Javed (2005), undersökte Advani & Gonsai (2019) krypteringsalgoritmerna AES, DES, 3DES och Blowfish. Utöver krypteringstid mätte de även dekrypteringstid. Författarna konstaterade att AES och Blowfish är snabbast, med likvärdig krypterings- och dekrypteringstid. Studien visar även att 3DES är långsammast av alla testade algoritmer.

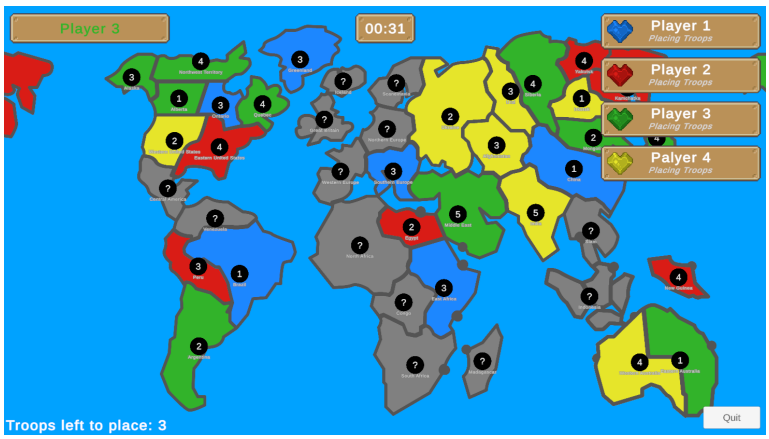
Panda (2016) har mätt krypteringstid, dekrypteringstid och *throughput* för både symmetriska och asymmetriska krypteringsalgoritmer. *Throughput* beskriver hur mycket data som algoritmen kan kryptera eller dekryptera per tidsenhet och beräknas genom att ta storleken på data som krypteras eller dekrypteras delat med den tid det tar för algoritmen att behandla given data. Högre *throughput* är bättre (Panda, 2016; Ramesh & Suruliandi, 2013). AES, DES, Blowfish och RSA undersöktes. Panda (2016) konstaterade att AES har bäst prestanda baserat på de faktorer som mätts.

Utöver exekveringstid och *throughput* har Ramesh & Suruliandi (2013) även undersökt minnesanvändning för AES, DES och Blowfish. Författarna drar slutsatsen att Blowfish presterar bäst av de tre testade algoritmerna, då den har lägst exekveringstid och minnesanvändning samtidigt som den har högst *throughput*. AES har sämst *throughput* och exekveringstid, men näst bäst minnesanvändning. Författarna nämner dock att AES kan vara lämplig för användning i spelapplikationer. Likt Ramesh & Suruliandi (2013), undersökte Panda & Nag (2015) bland annat minnesanvändning för AES, Blowfish och Salsa20. Resultatet visar att AES är snabbast av algoritmerna, men den har också högst minnesanvändning, medan Salsa20 använder minst minne.

Routray m. fl. (2021) har undersökt krypteringstid, dekrypteringstid, *throughput* och responstid[2] för kombinationer av den asymmetriska krypteringsalgoritmen RSA med sex olika symmetriska algoritmer, AES, DES, 3DES, Blowfish, RC4 och Rabbit. Syftet med studien var att undersöka prestandan för krypteringsalgoritmer som har kombinerats för att stärka säkerheten. Författarna konstaterade att en kombination av RSA och Blowfish gav bäst resultat.

2.3 Spelprototyp

I denna studie har en egenutvecklad spelprototyp använts som testmiljö för att genomföra testerna i ett verkligt scenario. Den data som krypteras är spelarnas truppattacker. Spelprototypen baseras på spelet *RISK: Global Domination* (2020), med skillnaden att alla spelare utför sina drag samtidigt. Spelprototypen tillåter flera spelare att ansluta via Unity Relay (Unity Technologies, 2024b). En spelare agerar som värd-klient, vilket innebär att de kör både en klientinstans och en serverinstans. De andra spelarna ansluter till värden som klienter. Spelare kan sedan utföra trupplacering och truppattacker som skickas till värd-klienten där de evalueras. De resulterande förändringarna av kartan skickas tillbaka till klienterna.



Figur 3: Bild från den egenskapade spelprototypen

[2]Routray m. fl. (2021) utför sina tester i en klient-server struktur över ett nätverk och definierar därmed responstid som tiden det tar för en klients förfrågan att besvaras av servern.

3 Problemformulering

Kryptering är en viktig aspekt när det gäller kommunikation över nätverk, eftersom det förhindrar obehöriga från att ta del av informationen och försvårar modifikation av data som skickas (GeeksforGeeks, 2025b). Utan kryptering kan den skickade informationen modifieras. I spel är detta ett problem eftersom en användare kan använda externa program för att modifiera den skickade informationen och därigenom få ett övertag över andra spelare.

Användning av kryptering inom spel gör det möjligt att skicka information innan krypteringsnyckeln överförs. På så sätt kan en användare inte modifiera sitt drag efter att ha sett andra användares drag, eftersom krypteringsnyckeln för att dekryptera dessa inte skickas förrän användaren har skickat sitt eget drag. Det betyder att användare kan skicka sina drag till varandra utan risk för att andra användare använder informationen för att få ett övertag.

När data skickas över nätverket är det viktigt att beakta paketstorlek och den tid det tar att konstruera och dekonstruera paketen. Större paketstorlek ökar risken att paket förloras, eller gör att det tar lång tid att skicka. Om ett paket är för stort finns risken att det fragmenteras, set vill säga delas upp i flera mindre paket. Detta ökar i sin tur risken att ett paket förloras (Huston, 2024). Denna studie syftar till att undersöka exekveringstid och paketstorlek, då dessa faktorer är särskilt relevanta i spel. Begreppet exekveringstid används i detta arbete som en samlad term för både krypterings- och dekrypteringstid. Syftet är att få en översikt över hur olika krypteringsalgoritmer påverkar dessa faktorer och ge spelutvecklare en grund för att välja den mest lämpliga algoritmen för olika scenarier.

I detta arbete har kryptering inom datorspel undersökts, med fokus på kryptering av data som skickas över nätverk. Specifikt har arbetet inriktats på symmetriska krypteringsalgoritmer. Detta innebär att samma nyckel används för att kryptera och dekryptera data, till skillnad från asymmetriska krypteringsalgoritmer där en privat och en publik nyckel används. Symmetriska algoritmer är mer relevanta för denna studie eftersom de har snabbare exekveringstid, vilket passar bättre för spel (GeeksforGeeks, 2025a). Studien fokuserar på att mäta och jämföra paketstorlek samt tid för kryptering och dekryptering för de olika algoritmerna.

Frågeställning: Hur påverkar olika krypteringsalgoritmer datamängd, krypterings- och dekrypteringstid när data förbereds och skickas över ett nätverk i ett spelscenario?

3.1 Metodbeskrivning

Krypteringsalgoritmerna utvärderas genom ett systematiskt experiment i en testmiljö där de variabler som mäts inkluderar paketstorlek, samt krypterings- och dekrypteringstid. Tiden för paketöverföring mellan klient och server mäts inte, eftersom detta är starkt beroende av klienters uppkoppling och förväntas inte påverkas nämnvärt av den valda krypteringsmetoden. Det tas därför inte upp i denna studie.

Testmiljön består av en egenutvecklad spelprototyp baserad på spelet *RISK: Global Domination* (2020) och skapad i spelmotorn Unity (Unity Technologies, 2024a). Det specifika scenario som testas är när spelare skickar in truppattacker till servern. I scenariot skickar klienter först en krypterad version av sina speldrag för rundan till servern. När all klientdata mottagits skickar servern en begäran till klienterna om deras krypteringsnycklar. När nycklarna har mottagits av servern används de för att dekryptera paketen. Detta säkerställer att en tredje part inte kan få tillgång till information om innehållet i paketen innan alla klienters data har nått servern, vilket förhindrar att de ändrar sitt drag. De komponenter av spelprototypen som används omfattar främst datastrukturerna för truppattacker och spelkartan.

Följande faktorer mäts under testet:

1. Paketstorlek
2. Tid för kryptering
3. Tid för dekryptering

Följande krypteringsalgoritmer testas:

1. Advanced Encryption Standard (AES)
2. Data Encryption Standard (DES)
3. Triple Data Encryption Standard (3DES)
4. Blowfish
5. Twofish

För AES, DES och 3DES har implementationer från både .NET (Microsoft Learn, 2025b) och Bouncy Castle (2025) använts. Implementationer av Blowfish och Twofish har använts från Bouncy Castle (2025). För AES och Twofish har tre olika tester utförts med nyckelstorlekar på 128, 192, och 256 bitar. Blowfish har testats med nyckelstorlekar på 32 och 448 bitar, där dessa extremvärden valdes för att testa den största möjliga variationen i nyckelstorlek.

Testet upprepas 100 gånger för varje algoritm, för att extremvärden påverkar resultaten. Varje algoritm testas på olika problemstorlekar som består av truppförflyttningsdata. En truppförflyttningsorder består av ett spelar-ID (den spelare som utför ordern), två territorium-ID (de territorier som trupperna flyttas från och till), samt antalet trupper som förflyttas. Varje algoritm startar med noll truppförflyttningsorder, och i varje iteration av testet ökas antalet order med ett, upp till ett maximum av 100 order. Värdena i orderna genereras slumpmässigt baserat på ett förutbestämt *seed*, vilket garanterar att alla algoritmer testas på samma data. Eftersom tid för överföring av paket inte mäts, körs både servern och klienten på samma dator. Alla tester utförs separat på två olika datorer, och deras hårdvaruspecifikationer kan ses i Figur 4.

	Dator A	Dator B
Processor	AMD Ryzen 7 5800H	Intel Core i7-8700 CPU
Klockhastighet (GHz)	3,20	3,20
Kärnor (st)	8	6
Minne (GB)	16	16
Operativsystem	Windows 11	Windows 10

Figur 4: Översikt av hårdvaruspecifikationer

Mätvärdena från experimentet på de olika datorerna jämförs sedan. Mätvärdena för krypterings- och dekrypteringstid anges i millisekunder (ms), och mätvärden för paketstorlek anges i *bytes* (B). Stopwatch-klassen i C# (Microsoft Learn, 2025a) används för att mäta både krypterings- och dekrypteringstid. Tidtagningen startas omedelbart innan kryptering eller dekryptering påbörjas och stoppas direkt efter att processen är slutförd.

3.2 Metoddiskussion

I arbetet har paketstorlek efter kryptering valts att mätas för algoritmerna, eftersom mängden *bytes* som skickas påverkar nätverksbelastningen. Större paket leder till högre belastning på nätverket och servern som tar emot paketen. Tiden det tar att skicka ett paket påverkas även av dess storlek. Alla dessa faktorer påverkar både server- och uppkopplingskostnader. Utöver paketstorlek mäts även krypteringstid och dekrypteringstid eftersom det är högst relevant i spel där latens är en viktig faktor. Högre latens leder till en mindre responsiv upplevelse, vilket kan påverka spelaren negativt (Liu m. fl., 2021). Dessutom leder högre krypterings- eller dekrypteringstider till att utvecklarna har mindre tid varje bildruta att använda för andra operationer. Krypterings- och dekrypteringstid är inte särskilt relevanta i spelprototypen som används för testerna i denna studie, men det är tänkbart att dessa faktorer är relevanta för andra speltyper som behöver skicka data oftare. Därför är det intressant att undersöka dem.

Tiden det tar att skicka ett paket har valts att inte mätas, då den är beroende av uppkopplingshastigheten som kan variera över tid. Tiden det tar att skicka ett paket är också beroende av paketets storlek vilket redan mäts (Huston, 2024). Därför valdes det att inte mäta denna tid. Minnesanvändning för kryptering och dekryptering av ett paket har också valts att inte mätas, då det bedömdes som för svårt att mäta inom arbetets tidsram, även om det hade varit en intressant faktor att undersöka.

En spelprototyp inspirerad av *RISK: Global Domination* (2020), skapad under en tidigare kurs, valdes för att kunna testa krypteringsalgoritmerna i ett verkligt scenario. Ett nytt projekt, skapat specifikt för att testa algoritmerna, skulle ha kunnat användas, men detta hade minskat resultatens ekologiska validitet (Preece m. fl., 2016, s.582–583). Användningen av data genererad från en spelprototyp istället för helt slumpmässigt genererad data kan påverka resultatet, då vissa algoritmer kan fungera annorlunda på vissa typer av data, vilket bland annat Advani & Gonsai (2019) visar. Användning av andra spel, inte skapade av författar-

na, valdes också bort, då detta hade lagt till en extra nivå av komplexitet och varit betydligt mer tidskrävande. Spelprototypen är dessutom skapad med detta arbete i åtanke, vilket gör implementationen av algoritmerna enklare.

De algoritmer som undersöks i detta arbete bedöms vara de mest populära symmetriska krypteringsalgoritmerna då de benämns som vanliga av flertalet artiklar (Nadeem & Javed, 2005; Panda & Nag, 2015; Salama m. fl., 2011; Sanap & More, 2021). De valda algoritmerna förekommer dessutom i ytterligare artiklar, där det dock inte anges varför just dessa algoritmer valdes (Advani & Gonsai, 2019; Panda, 2016; Ramakrishna & Ali Shaik, 2025; Ramesh & Suruliandi, 2013; Routray m. fl., 2021; Yassein m. fl., 2017). AES anses vara den mest populära algoritmen, då den klassificeras som en standard av amerikanska myndigheter (National Institute of Standards and Technology (NIST) m. fl., 2023). Eftersom Unity (Unity Technologies, 2024a) använder .NET (Microsoft Corporation, 2018), finns redan flera algoritmer implementerade genom Cryptography Namespace i C# (Microsoft Learn, 2025b). Det är därför sannolikt att spelutvecklare väljer att använda någon av dessa algoritmer. Ytterligare algoritmer valdes från paketet Bouncy Castle (2025), som också är utvecklat för .NET. Detta säkerställer även att implementationerna av algoritmerna följer industristandard och redan är optimerade samt väl testade.

Metoden som används för att utföra testerna liknar den som används av Panda & Nag (2015), med skillnaden att de genomför tester i en applikation specifikt utvecklad för ändamålet, istället för i en spelprototyp. De mäter även minnesanvändning vilket inte görs i denna studie, utan istället mäts paketstorlek.

För varje algoritm och variation upprepas testet 100 gånger per problemstorlek, för att minimera effekten av extremvärden orsakade av externa variabler och för att beräkna ett medelvärde. Externa variabler kan exempelvis vara andra bakgrundsprocesser på datorn som använder resurser. Det kan resultera i en längre exekveringstid, då den uppmäta tiden inte enbart inkluderar exekvering av krypteringsalgoritmen. 100 upprepningar valdes, då detta bedömdes ge ett bra medelvärde, samtidigt som testernas genomförande inte tar för långt tid. Varje algoritm testas på problemstorlekar mellan noll och 100 truppattacker, eftersom vissa algoritmer är mer lämpade för mindre datamängder, medan andra passar bättre för större datamängder. Likt Nadeem & Javed (2005) och Ramesh & Suruliandi (2013) genomförs testerna på två olika datorer för att minimera hårdvaruspecifika skillnader som kan påverka exekveringstiden, såsom processorkraft och RAM-minne. Som kan ses i Figur 4, har de valda datorerna relativt lika hårdvaruspecifikationer. Detta beror på att det var dessa datorer som fanns tillgängliga för författarna.

Stopwatch-klassen i C# valdes, då den enligt Microsoft ”Provides a set of methods and properties that you can use to accurately measure elapsed time” (Microsoft Learn, 2025a). Dessutom tillhandahåller Stopwatch-klassen variabler för att mäta tidsmätningens precision på datorns specifika hårdvara, vilket gör det möjligt att beräkna felmarginaler. Detta är väl lämpat för arbetets tester. I Unity finns ett antal inbyggda metoder för tidsmätning, men dessa bedömdes mindre lämpliga, då ingen av dem tillhandahåller felmarginalsmätning.

Krypteringsalgoritmernas säkerhet undersöks inte i detta arbete, då detta bedömdes vara utanför arbetets omfattning. Författarna saknar dessutom tillräckliga kunskaper för att kunna utvärdera säkerheten hos en algoritm. Säkerhetsutvärdering är ett extremt komplext område, mer lämpat för matematiker och cybersäkerhetsexperter. Säkerhet är dock en viktig faktor vid val av algoritm, och därför används tidigare arbeten för att ge en översikt av algoritmernas säkerhet.

4 Genomförande

Detta kapitel presenterar hur arbetet har utförts i detalj. Svårigheter som dykt upp under arbetet tas upp, såsom hur extremvärden har hanterats. Resultatet presenteras även, följt av en analys för att visa på intressanta mönster som kan observeras.

4.1 Utformning av testmiljö

Testmiljön är baserad på en egenutvecklad spelprototyp skapad i Unity (Unity Technologies, 2024a). Testerna utförs inte över ett nätverk eftersom inget av värdena som mäts kan påverkas av detta, utan både kryptering och dekryptering sker på samma enhet. Testerna utförs i en minimal testmiljö utan någon typ av visualisering för spelet, endast text över testernas framsteg visas på skärmen.

Först genereras truppattacker som ska krypteras, dessa består av ett startterritorium, ett närliggande territorium, antalet trupper som flyttas och vilken spelare som skickar kommandot. Spelar-ID är alltid satt till noll, medan antalet trupper slumpas mellan 1 och 99. Territorier väljs slumpmässigt bland de tillgängliga på spelkartan, se Appendix A. Alla slumpmässiga värden genereras utifrån ett *seed*[3], vilket återställs inför varje algoritm. Detta säkerställer att samtliga algoritmer testas på identisk data.

Därefter genomförs först kryptering följt av dekryptering av den data som genererats med den valda algoritmen. AES, DES och 3DES är implementerade med hjälp av Cryptography Namespace i C# (Microsoft Learn, 2025b), se Appendix B. AES, DES och 3DES har också implementerats med hjälp av paketet Bouncy Castle (2025) tillsammans med Blowfish och Twofish, se Appendix C. Efter varje upprepning sparas krypteringstid, dekrypteringstid, samt datans storlek i bytes innan och efter kryptering, för att möjliggöra bortsortering av extremvärden.

4.2 Resultat

Efter att experimenten genomförts och mätvärden samlats in, visualiserades resultaten i ett antal grafer. Syftet med detta avsnitt är endast att presentera den data som erhållits från experimenten. Analys av data sker först i nästa kapitel.

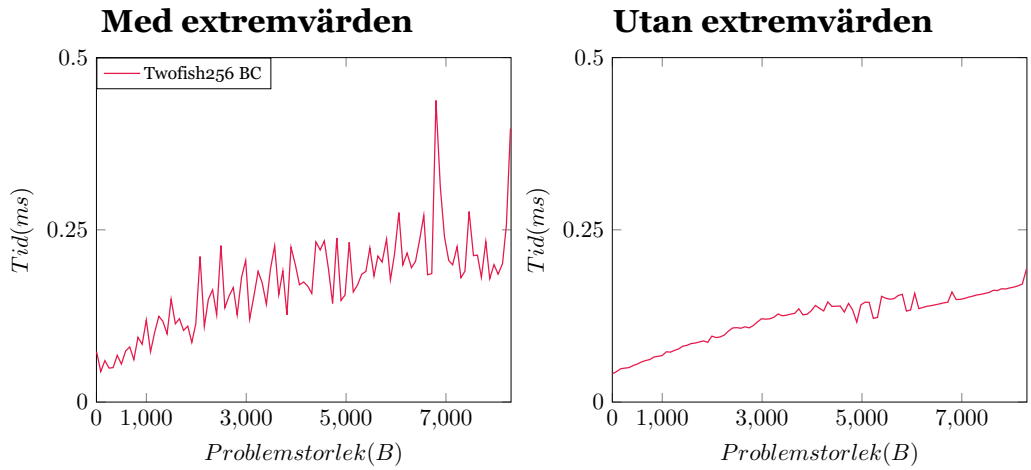
4.2.1 Extremvärden

Extremvärden i exekveringstid exkluderades med hjälp av interkvartilintervall. Datasetet delas in i fyra lika stora delar av tre kvartiler. 25% av dataset är mindre än Q_1 och 75% är större. Q_3 är det motsatta, 75% av datasetet är mindre och 25% är större. Interkvartilintervallet beräknas $IQR = Q_3 - Q_1$ och beskriver spridningen i datasetet. Alla värden som faller utanför det definierade intervallet i Ekvation 1 exkluderas (Benkert m. fl., 2008; Dekking m. fl., 2005, s.234–236).

[3] Ett *seed* används för att initialisera en pseudoslumptalsgenerator så att den alltid genererar samma sekvens av värden.

$$Q_1 - 1,5 \cdot IQR < n < Q_3 + 1,5 \cdot IQR \tag{1}$$

Extremvärden exkluderas eftersom de visade sig ha en betydande påverkan på resultatet. På grund av de korta tidsintervallen som mäts kan externa faktorer ha en oproportionerligt stor inverkan på resultatet. I vissa fall observerades extremvärden som var upp till 20 gånger högre än majoriteten av mätvärdena inom samma experimentiteration. Detta ledde till ett icke-representativt medelvärde.

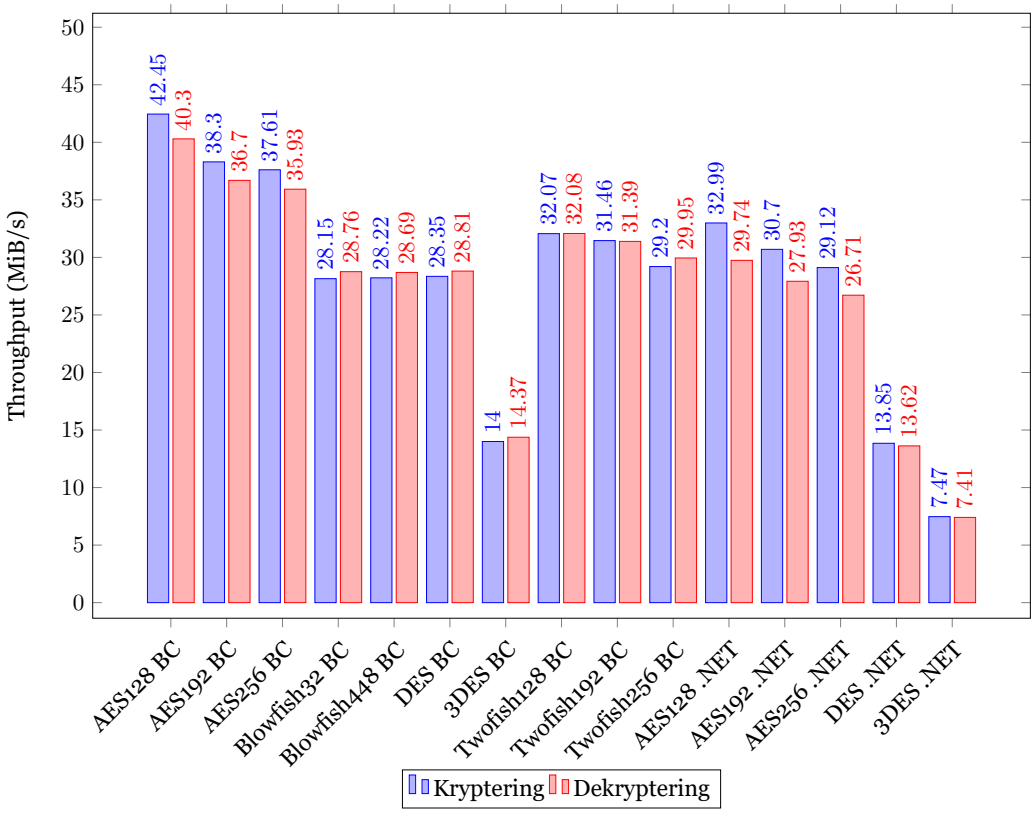


Figur 5: Jämförelse med och utan extremvärden

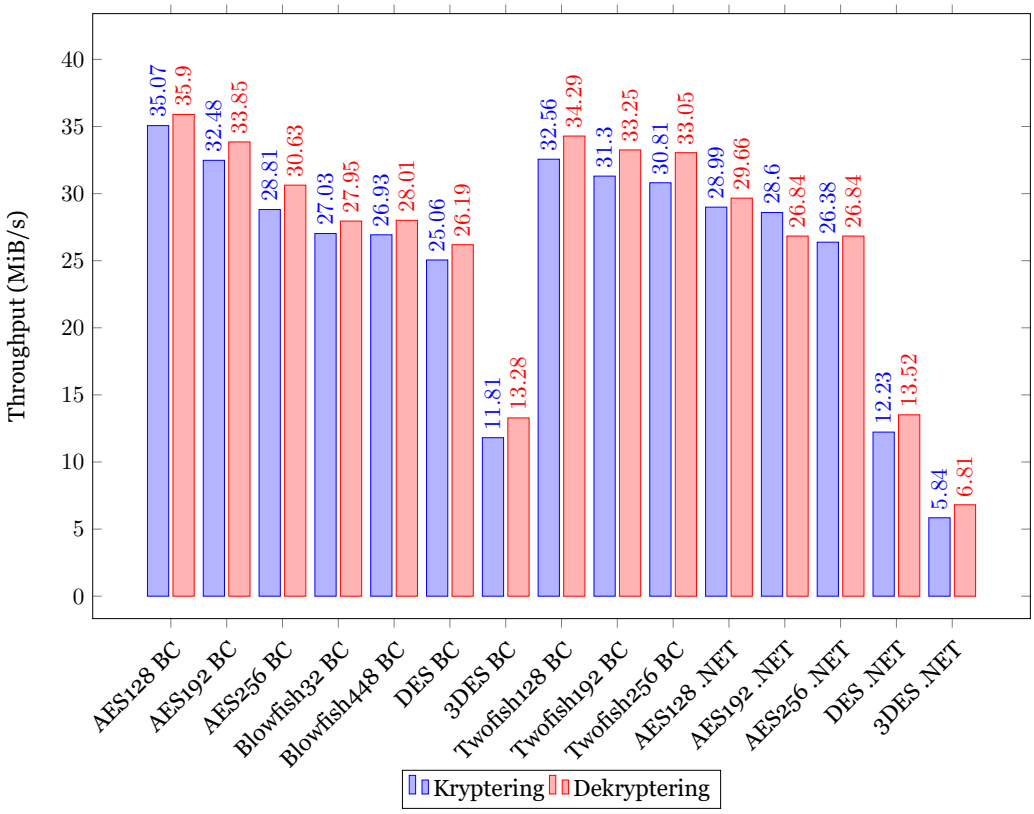
I Figur 5 visas ett exempel för hur processen att ta bort extremvärden påverkar den resulterande datan. Där kan det ses att grafens övergripande form behålls, samtidigt som stora avvikelser i datan elimineras. De exkluderade datapunkterna var vanligtvis 3-5 gånger högre än närliggande värden. Denna process hade en liknande effekt på samtliga dataset, vilket framgår i Appendix D jämfört med Appendix E.

4.2.2 Exekveringstid

Algoritmernas exekveringstid presenteras i form av *throughput*, vilket underlättar visualiseringen. *Throughput* beskriver mängden data som kan krypteras per tidsenhet. *Throughput* presenteras i Figur 6 för dator A och Figur 7 för dator B, uttryckt i enheten mebibytes per sekund (MiB/s). Dessa diagram representerar en sammanställning av alla resultat som utvunnits från samtliga tester. Nyckelstorleken för algoritmerna anges genom siffran efter namnet. Exempelvis AES128, där nyckelstorleken är 128 bitar. Vilken implementation som används indikeras med BC för Bouncy Castle eller .NET för de implementationer från Cryptography Namespace i C#. Algoritmernas individuella krypterings- och dekrypteringstid uppdelat efter dator kan ses i Appendix D.



Figur 6: *Throughput* för dator A



Figur 7: *Throughput* för dator B

4.2.3 Paketstorlek

Denna studie undersöker även ökningen av paketstorlek efter kryptering, vilket presenteras i Figur 8. Eftersom algoritmerna är deterministiska och den initiala datan är identisk, redovisas resultatet inte separat för de olika datorerna eller olika nyckelstorlekar, då dessa faktorer inte påverkar resultatet. Eftersom resultatet är identiskt för implementationerna av algoritmerna i både .NET och Bouncy Castle, presenteras dessa inte separat.

Figur 8 visar den genomsnittliga ökningen av paketstorleken, det vill säga den slutgiltiga storleken efter kryptering. Den genomsnittliga ökningen är konstant och oberoende av storleken på indata.

Algoritm	Blockstorlek (B)	Genomsnittlig ökning (B)	Min ökning (B)	Max ökning (B)
AES	16	8,90	1	16
Twofish	16	8,90	1	16
DES	8	4,42	1	8
3DES	8	4,42	1	8
Blowfish	8	4,42	1	8

Figur 8: Ökning i paketstorlek

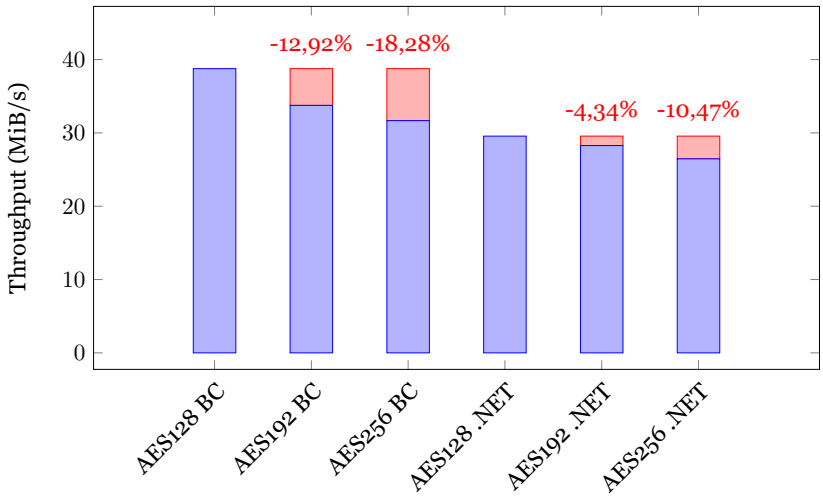
4.3 Analys

Detta stycke presenterar en analys av resultatet. Precis som förra stycket har analysen av exekveringstid och paketstorlek delats in i separata stycken.

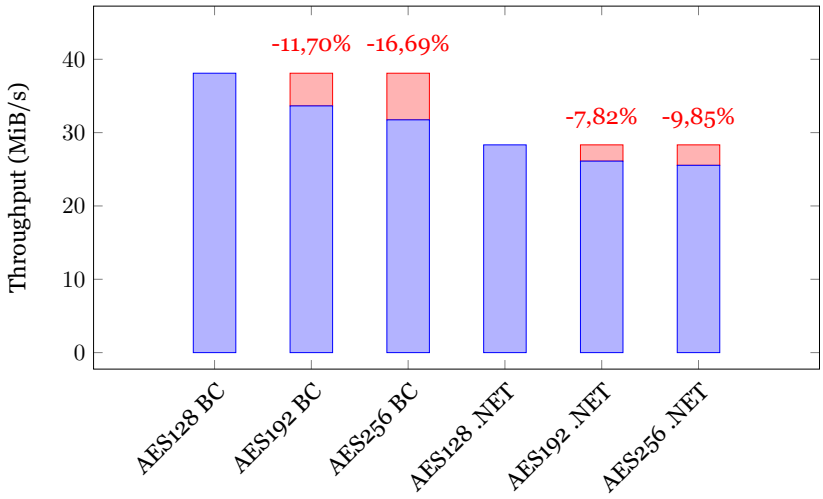
4.3.1 Exekveringstid

Utifrån Figur 6 och Figur 7 kan några faktorer som påverkar exekveringstiden identifieras. Exekveringstiden påverkades märkbart av antalet rundor som algoritmen genomför, vilket framgår av testerna med AES. AES128, som genomför tio rundor, uppvisar kortare krypterings- och dekrypteringstid jämfört med AES192 och AES256, vilka genomför tolv respektive 14 rundor.

För implementationerna från Bouncy Castle har AES192 och AES256 i genomsnitt **12,92%** respektive **18,28%** mindre krypterings-*throughput* och **11,70%** respektive **16,69%** mindre dekrypterings-*throughput* än AES128. Motsvarande värden för implementationerna från .NET är **4,34%** respektive **10,47%** mindre krypterings-*throughput* och **7,82%** respektive **9,85%** mindre dekrypterings-*throughput*, se Figur 9 och Figur 10.



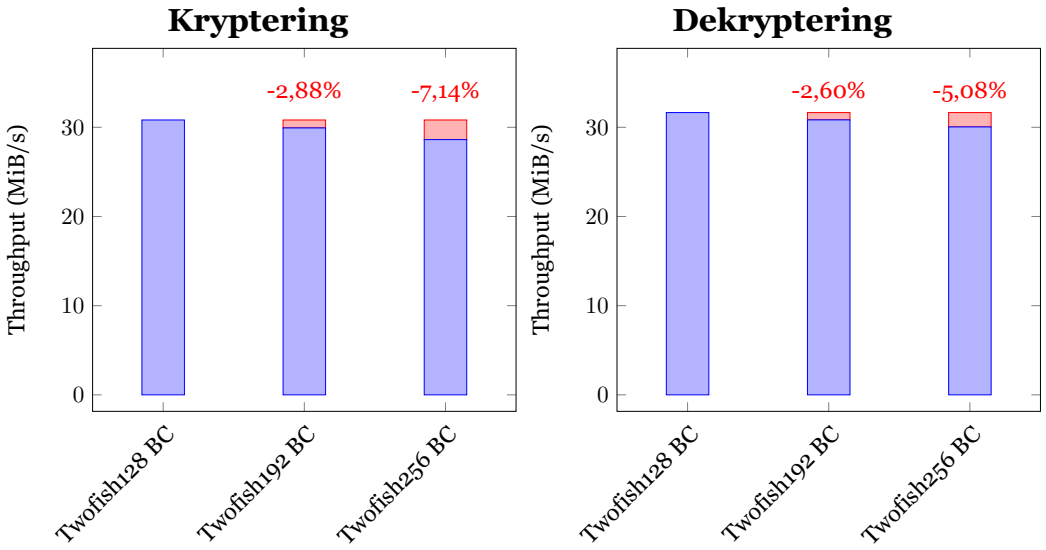
Figur 9: Minskning i krypterings-*throughput* för AES



Figur 10: Minskning i dekrypterings-*throughput* för AES

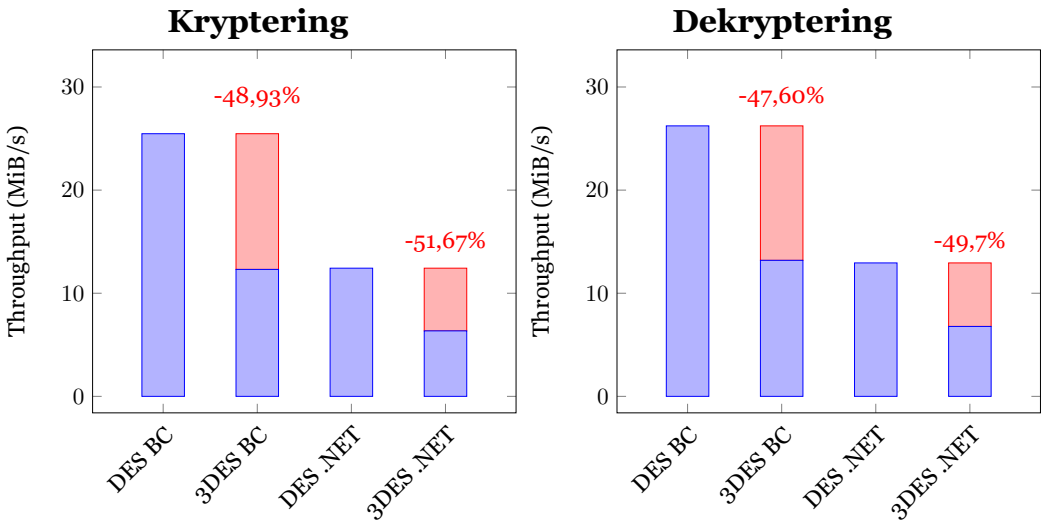
Det framgår även att antalet rundor har en mindre inverkan på exekveringstiden i .NET jämfört med Bouncy Castle. Däremot är .NET i genomsnitt **18,81%** respektive **22,52%** långsammare än Bouncy Castle för kryptering och dekryptering.

Mätvärdena från Twofish visar på att storleken på krypteringsnyckeln kan påverka exekveringstiden, dock inte i lika hög grad som en ökning i antal rundor som körs. Twofish192 och Twofish256 har **2,88%** respektive **7,14%** lägre krypterings-*throughput* och **2,60%** respektive **5,08%** mindre dekrypterings-*throughput* än Twofish128, se Figur 11. Däremot påverkas inte exekveringstiden av krypteringsnyckelns storlek i de två testerna med Blowfish.



Figur 11: Minskning i krypterings- och dekrypterings-*throughput* för Twofish

3DES var mycket långsammare än DES, dock inte tre gånger långsammare. För implementationerna från Bouncy Castle hade 3DES **48,93%** respektive **47,60%** lägre krypterings- och dekrypterings-*throughput* än DES. För implementationerna från .NET var dessa värden istället **51,67%** respektive **49,7%**, se Figur 12. Implementationerna från Bouncy Castle var **103,75%** respektive **94,18%** snabbare för DES och 3DES än sina motsvarigheter i .NET.



Figur 12: Minskning i krypterings- och dekrypterings-*throughput* för DES jämfört med 3DES

Som ses i Figur 6 är kryptering överlag snabbare än dekryptering för dator A. Dock är det motsatta sant för dator B där dekryptering överlag är snabbare än kryptering, se Figur 7. Detta indikerar att krypterings- och dekrypteringstiderna kan variera i relation till varandra beroende på hårdvara. Däremot är det oklart vilka faktorer som orsakar detta.

Utöver tidsökningen som observerades i varje iteration syns också en initial kostnad hos algoritmerna. Från graferna i Appendix D kan det ses att alla algoritmer har varierande men liknande initial kostnad. Detta skulle kunna påverka beräkningen av *throughput*, men eftersom alla algoritmer har ungefär samma initial kostnad bedöms det inte som tillräckligt signifikant för att påverka resten av resultatet. Den initiala kostnaden är att förvänta sig eftersom det är en vanlig förekomst hos algoritmer, kostnaden kan bestå av exempelvis minnesallokering. Eftersom det inte är känt exakt hur implementationerna som används fungerar kan inga direkta slutsatser dras om vad som påverkar i detta fall.

4.3.2 Paketstorlek

Den genomsnittliga ökningen av paketstorleken var ungefär hälften av blockstorleken för alla algoritmer, oberoende på storleken av indata. Detta innebär en genomsnittlig ökning på **8,90** *bytes* för AES och Twofish, samt en ökning på **4,42** *bytes* för DES, 3DES och Blowfish som kan ses i Figur 8. Anledningen till ökningen är att algoritmerna krypterar block av data, och måste då nödvändigtvis öka den krypterade storleken till närmaste större multipel av blockstorleken, för att få plats med all data. Det sista blocket kommer därför endast delvis innehålla användbar data och delvis tom data.

4.4 Slutsats

Krypterings- och dekrypteringstid var beroende av både antal rundor som algoritmen körde, och i vissa fall storleken på krypteringsnyckeln. Att antalet rundor påverkar exekveringstiden syns tydligast på AES, där en stor ökning i exekveringstid kunde observeras mellan olika tester, både i Bouncy Castle och .NET. Minskningen i *throughput* var i vissa fall så stor som **18,28%**. Alla andra testade krypteringsalgoritmer kör en konstant mängd rundor oberoende av nyckelstorlek och påverkas därför inte av denna faktor. Krypteringsnyckelns storlek kan också påverka exekveringstiden, detta observerades dock endast för Twofish och inte för någon av de andra testade algoritmerna. Antalet rundor som algoritmerna kör visade sig dock vara den största påverkande faktorn när det gäller exekveringstid.

3DES är klart långsammast av alla algoritmer, följt av DES och Blowfish som har likvärdig prestanda för implementationerna från Bouncy Castle. AES var i de flesta fall snabbare än alla andra algoritmer, förutom i vissa fall för större nyckelstorlekar där Twofish istället var snabbare. Detta var däremot beroende på vilken dator som testet hade utförts på.

Den påverkande faktorn när det gäller ökning av paketstorlek var blockstorleken. Paketstorleken ökar i genomsnitt med lite över halva blockstorleken.

Vilken dator testerna kördes på påverkade också resultatet. Exempelvis var kryptering långsammare än dekryptering för dator B, samtidigt som kryptering var snabbare i majoriteten av fall för dator A. Vilken implementation det var som användes gjorde också stor skillnad, då det visade sig att implementationerna från .NET var betydligt långsammare än de från Bouncy Castle.

5 Sammanfattning och diskussion

Detta stycke syftar att sammanfatta hela arbetet, för att sedan presentera övriga tankar och idéer om arbetet och hur det förhåller sig till bakgrunden. Förslag på framtida arbete presenteras även, samt samhälleliga och etiska aspekter relaterade till arbetet.

5.1 Sammanfattning

Arbetet undersöker olika krypteringsalgoritmers påverkan på datamängd, krypterings- och dekrypteringstid när data förbereds för att skickas över ett nätverk. Detta har gjorts med hjälp av en egenutvecklad spelprototyp i Unity (Unity Technologies, 2024a), inspirerad av spelet *RISK: Global Domination* (2020). Algoritmerna AES, DES, 3DES, Blowfish och Twofish har undersökts med fokus på algoritmernas krypterings-, dekrypteringstider och storleksökning efter kryptering. För AES, DES och 3DES har två olika implementationer av algoritmerna testats, en från Cryptography Namespace i C# (Microsoft Learn, 2025b) och den andra från paketet Bouncy Castle (2025). Implementationerna för Blowfish och Twofish är tagna från Bouncy Castle.

Metoden som använts är ett systematiskt experiment där varje algoritm testats för olika problemstorlekar. Testerna är utförda i en egenutvecklad spelprototyp liknande spelet *RISK: Global Domination* (2020), för att ge ett mer realistiskt scenario. Den data som krypteras består av truppattacker bestående av ett spelar-ID, två territorium-ID:n och antalet trupper som förflyttas, slumpmässigt genererade genom spelprototypen.

Resultatet visar att det finns stora skillnader beroende på vilken krypteringsalgoritm som används, och att dessa kan kopplas till de faktorer som undersöktes. Blockstorleken visade sig huvudsakligen påverka paketstorleken, men resulterade inte i särskilt stora ökningar i paketstorlek. Antalet rundor som algoritmerna kör visade sig vara den huvudsakliga faktor som påverkade exekveringstiden, men även storleken på krypteringsnyckeln påverkade exekveringstiden för vissa algoritmer. Vilken implementation som användes visade sig även ha stor inverkan på resultatet, där de algoritmer som användes från .NET var långsammare än de från Bouncy Castle. Överlag bedöms det att AES presterade bäst över alla mätta faktorer, då båda de AES implementationer som testades överlag var snabbare än alla andra algoritmer. Det bedömdes även att DES och 3DES presterade sämre än alla andra algoritmer.

5.2 Diskussion

I detta arbete visade sig AES och Twofish vara de snabbaste av alla algoritmer följt av Blowfish, DES och till sist 3DES. Detta skiljer sig från Nadeem & Javed (2005), som konstaterade att Blowfish var snabbast följt av DES och till sist AES och 3DES som kom på delad sista plats. Deras studie utfördes dock 2005, och datorhårdvara och algoritmimplementationer har ändrats mycket under de senaste 20 åren. Även Ramesh & Suruliandi (2013) får ett resultat som skiljer sig från resultaten i denna studie. Författarna konstaterar att Blowfish är snabbast, följt av DES och sist AES. Studien utfördes 2013 och använde sig av äldre hårdva-

ra. Detta arbete visar att moderna implementationer av AES på modern hårdvara presterar betydligt bättre i jämförelse med DES och 3DES.

Även om datorhårdvara har förbättrats mycket under de senaste 20 åren, beror förmodligen majoriteten av förbättringarna av AES på bättre implementationer av algoritmen. Guo m. fl. (2015) visar bland annat att en implementation författarna valt att kalla ”AES *fast*” är cirka 50 gånger snabbare än en standardimplementation av AES.

Resultatet i detta arbete stämmer bättre överens med resultatet i Advani & Gonsai (2019), som visade att AES och Blowfish hade bäst prestanda, följt av DES och till sist 3DES. I deras studie visade sig AES vara likvärdig med Blowfish, vilket inte riktigt stämmer överens med resultatet i denna studie. Blowfish var dock betydligt bättre än DES och 3DES i detta arbete, vilket också Advani & Gonsai (2019) konstaterade. Resultatet i Panda & Nag (2015) stämmer också väl överens med detta arbete. Författarna konstaterade att AES var betydligt snabbare än Blowfish vilket även detta arbete visade.

Detta arbete är dock inte utan brister. En potentiell felkälla är de två datorerna som testerna har utförts på. Deras hårdvara är relativt lika varandras. Även om det fanns vissa skillnader i resultat mellan datorerna skulle dessa skillnader kunna vara betydligt större på mer annorlunda hårdvara. Resultatet från detta arbete är därmed beroende av hårdvaran som testerna utfördes på, då ett helt annat resultat hade kunnat uppnås på annan hårdvara.

Eftersom externa algoritmimplementationer har använts är det inte känt exakt hur implementationerna faktiskt fungerar. Det är oklart om de är implementerade på ett sådant sätt att det ger en realistisk bild av hur algoritmerna fungerar i relation till varandra. Eftersom det är relativt populära implementationer som har använts, kan det antas att detta är fallet, att de är rimligt väl implementerade, men det går inte att garantera utan en djupare blick in i implementationerna. En alternativ metod hade kunnat vara att skapa egna implementationer av algoritmerna, men även denna metod har sina brister.

En faktor som skulle kunna ha påverkat körningen av testerna är de bakgrundsprocesser som körs på datorerna. Båda datorerna som har använts kör Windows, vilket innebär att en del processer hela tiden är aktiva i bakgrunden som skulle kunna påverka prestandan. Eftersom personliga datorer har använts, alltså datorer som inte var speciellt uppsatta för dessa tester, finns det också risk för att användare har installerat programvara som körs i bakgrunden och påverkar resultatet, exempelvis genom att ta upp resurser eller introducera tillfälliga spikar i prestanda. Dessa faktorer skulle kunna uppenbara sig i mätdata genom en generell sänkning av prestanda eller extremvärden. Som beskrevs tidigare har viss tid lagts på att sortera bort extremvärden, men dessa metoder är inte helt utan brister.

En aspekt som inte har undersökts i detta arbete är säkerheten hos algoritmerna. Ramakrishna & Ali Shaik (2025) undersöker säkerheten hos flera krypteringsalgoritmer, inklusive de som används i detta arbete. De bedömde att AES var säkrast, följt tätt av Blowfish och Twofish. DES bedömdes som klart sämst, då de pekade ut att algoritmen var känslig för *brute-*

force attacker, på grund av dess korta nyckelstorlek. Dessa säkerhetsproblem förbättrades med utvecklingen av 3DES, men med nackdelen av längre exekveringstid, vilket även visade sig i de tester som utförts i detta arbete. Ramakrishna pekar på krypteringshastighet som en av fördelarna med både Blowfish och Twofish, vilket även märktes i detta arbete. Sanap & More (2021) har också undersökt AES, DES och 3DES ur ett säkerhetsperspektiv. Genom mätning av entropi, *floating frequency* och histogramanalys drar författarna slutsatsen att AES har bäst säkerhet av de undersökta algoritmerna.

Skillnaden i exekveringstid mellan 3DES, som var den långsammaste algoritmen och AES som var den snabbaste algoritmen var stor. AES var upp till fyra gånger så snabb som 3DES, beroende på implementation. Enligt Madanapalli m. fl. (2022) kräver de flesta online spel endast ett fåtal hundra kilobit per sekund (kb/s). Om ett spel kräver 300 kb/s och spelutvecklaren vill uppnå 60 bildrutor per sekund, innebär detta att skillnaden i exekveringstid mellan 3DES och AES endast blir 0,07 ms per bildruta[4]. Detta visar att även om AES är betydligt snabbare än 3DES, är krypteringsalgoritmens exekveringstid låg i jämförelse med de 16,7 ms som finns tillgängliga för en bildruta.

Resultatet från detta arbete är inte bara applicerbart i ett spelscenario utan kan även användas i andra typer av mjukvara såsom kommunikationsapplikationer där snabbare kryptering leder till snabbare kommunikation. Exekveringstid är extra viktigt för realtidskommunikation, som till exempel videosamtal. Resultatet från detta arbete kan då användas som en utgångspunkt vid utvecklingen av sådan programvara.

Ökningen av paketstorlek efter kryptering visade sig inte vara en stor faktor, eftersom den resulterande storleken som mest ökar med en blockstorlek. Detta gör att det endast skulle vara relevant när den okrypterade datamängdens storlek är mycket liten. I fall där den okrypterade datamängden är större, blir ökningen alltså försumbar.

5.3 Samhälleliga och etiska aspekter

Användningen av krypteringsalgoritmer är bland annat viktig för att skydda individers integritet på nätet. Detta kan vara i form av att kryptera känsliga meddelanden, lösenord eller annan konfidentiell information. Att säkerställa både säkerheten och effektiviteten hos de använda krypteringsalgoritmer är därför fundamentalt för att skapa en säker miljö på nätet både för individer och företag. Det är därför viktigt att fortsatt forskning bedrivs kring säkerheten hos använda krypteringsalgoritmer för att identifiera möjliga brister och utveckla lösningar till dessa.

Det finns även nackdelar med kryptering som måste tas i beaktande, exempelvis användningen av krypterade meddelandekanaler i syfte att planera och utföra olika brottsliga aktiviteter. Kryptering kan därmed försvåra polisens arbete med att förhindra och utreda brott.

[4] Denna beräkning förutsätter att den data som skickas är jämnt fördelad över alla bildrutor.

Energiförbrukning är en viktig aspekt i dagens samhälle. Lägre energiförbrukning är mer ekonomiskt och bättre för miljön. Salama m. fl. (2011) har undersökt energiförbrukning hos sex symmetriska krypteringsalgoritmer som de beskriver som de mest populära. Dessa algoritmer är AES, DES, 3DES, Blowfish, RC2 och RC6. Studien visar att Blowfish och RC6 har lägst energiförbrukning för kryptering av textfiler och ljudfiler. Dock har de, tillsammans med RC2, den högsta energiförbrukningen av alla algoritmer vid kryptering av bildfiler. DES använder då minst energi följt av 3DES och AES.

5.4 Framtida arbete

Detta arbete har främst fokuserat på att undersöka specifika symmetriska krypteringsalgoritmer. Framtida arbete skulle kunna fokusera på att undersöka algoritmer som detta arbete inte tar upp. Det hade även varit intressant att undersöka osymmetriska algoritmer, och de skillnader som finns mellan dem. Även intressant hade varit att undersöka algoritmerna i fler spelmotorer än Unity, för att se om det finns någon bias mot specifika algoritmer från spelmotorns sida. Detta skulle även möjliggöra tester av fler implementationer av algoritmerna i andra programmeringsspråk, exempelvis om tester utförs i Unreal Engine skulle implementationer skrivna i C++ kunna användas.

En annan intressant aspekt att undersöka är minnesanvändning hos de olika algoritmerna. Detta skulle vara högst relevant vid kryptering av större mängder data, exempelvis vid kryptering av bilder eller videor. Det vore även intressant att utforska hur olika algoritmer presterar på olika sorters data, om vissa är snabbare eller långsammare på att exempelvis kryptera bilder.

Resultatet i detta arbete visade att det fanns en stor skillnad i krypterings- och dekrypteringstid beroende på vilken dator algoritmerna testades på. Dock kommer detta arbete inte fram till varför denna skillnad uppstår. Framtida arbete skulle därför kunna fokusera på att genomföra fler tester på olika datorer, med fokus på att komma fram till varför dessa skillnader uppstår.

I detta arbete har säkerheten hos algoritmerna inte legat i särskilt stort fokus. Framtida arbete skulle kunna fokusera på dessa aspekter, och utforska vilka faktorer som gör vissa algoritmer säkrare än andra, eller vilken sorts attacker de är känsliga mot. En relevant aspekt att utforska i detta sammanhang är huruvida olika typer av data påverkar algoritmers säkerhet, exempelvis om användningen av bilder introducerar mönster som kan göra algoritmen mer sårbar.

Referenser

Advani, N. A. & Gonsai, A. M. (2019). Performance Analysis of Symmetric Encryption Algorithms for their Encryption and Decryption Time. I: *2019 6th International Conference on Computing for Sustainable Global Development (INDIACom)*. New Delhi, Indien 13-15 March 2019, s. 359–362.

Benkert, K., Gabriel, E. & Resch, M. M. (2008). Outlier detection in performance data of parallel applications. I: *2008 IEEE International Symposium on Parallel and Distributed Processing*. Miami, USA 14-18 April 2008, s. 1–8. DOI: 10.1109/IPDPS.2008.4536463.

Bouncy Castle (2025). *Bouncy Castle open-source cryptographic APIs*. Version 2.5.0. URL: https://www.bouncycastle.org/.

Coppersmith, D., Holloway, C., Matyas, S. M. & Zunic, N. (1997). The data encryption standard. I: *Information Security Technical Report* 2(2), s. 22–24. DOI: https://doi.org/10.1016/S1363-4127(97)81325-8.

Dekking, F. M., Kraaikamp, C., Lopuhaä, H. P. & Meester, L. E. (2005). *A Modern Introduction to Probability and Statistics*. Springer London. DOI: 10.1007/1-84628-168-7.

Electronic Frontier Foundation (EFF) (1998). *Frequently Asked Questions (FAQ) About the Electronic Frontier Foundation's "DES Cracker" Machine*. URL: https://w2.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/HTML/19980716_eff_des_faq.html (hämtad 2025-02-25).

GeeksforGeeks (2023a). *Block Cipher Design Principles*. Section: Computer Networks. URL: https://www.geeksforgeeks.org/block-cipher-design-principles/ (hämtad 2025-02-25).

GeeksforGeeks (2023b). *Data encryption standard (DES) | Set 1*. URL: https://www.geeksforgeeks.org/data-encryption-standard-des-set-1/ (hämtad 2025-02-19).

GeeksforGeeks (2024). *Twofish Encryption Algorithm*. URL: https://www.geeksforgeeks.org/twofish-encryption-algorithm/ (hämtad 2025-02-10).

GeeksforGeeks (2025a). *Difference Between Symmetric and Asymmetric Key Encryption*. URL: https://www.geeksforgeeks.org/difference-between-symmetric-and-asymmetric-key-encryption/ (hämtad 2025-01-27).

GeeksforGeeks (2025b). *What is Data Encryption?* URL: https://www.geeksforgeeks.org/what-is-data-encryption/ (hämtad 2025-02-05).

Guo, G.-l., Qian, Q. & Zhang, R. (2015). Different Implementations of AES Cryptographic Algorithm. I: *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. New York, USA 24-26 Augusti 2015, s. 1848–1853. DOI: 10.1109/HPCC-CSS-ICESS.2015.215.

Huston, G. (2024). *The Size of Packets | blabs*. URL: https://labs.apnic.net/index.php/2024/10/04/the-size-of-packets/ (hämtad 2025-01-27).

Irdeto (2018). *Irdeto Global Gaming Survey: The last checkpoint for cheating*. URL: https://irdeto.com/hubfs/resources/reports/global-gaming-report-2018.pdf (hämtad 2025-02-10).

Liu, S., Claypool, M., Kuwahara, A., Scovell, J. & Sherman, J. (2021). The Effects of Network Latency on Competitive First-Person Shooter Game Players. I: *2021 13th International Conference on Quality of Multimedia Experience (QoMEX)*. Montréal, Kanada 14-17 Juni 2021, s. 151–156. DOI: 10.1109/QoMEX51781.2021.9465419.

Madanapalli, S. C., Gharakheili, H. H. & Sivaraman, V. (2022). Know Thy Lag: In-Network Game Detection and Latency Measurement. I: *Passive and Active Measurement*. Virtuellt Evenemang 28–30 Mars 2022, s. 395–410. DOI: 10.1007/978-3-030-98785-5_17.

Merkle, R. C. & Hellman, M. E. (1981). On the security of multiple encryption. I: *Commun. ACM* 24(7), s. 465–467. DOI: 10.1145/358699.358718.

Microsoft Corporation (2018). *Microsoft .NET Standard*. Version 2.1. URL: https://learn.microsoft.com/en-us/dotnet/standard/net-standard?tabs=net-standard-2-1.

Microsoft Learn (2025a). *Stopwatch Class (System.Diagnostics)*. URL: https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?view=net-9.0 (hämtad 2025-01-27).

Microsoft Learn (2025b). *System.Security.Cryptography Namespace*. URL: https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography?view=net-9.0 (hämtad 2025-01-23).

Nadeem, A. & Javed, M. (2005). A Performance Comparison of Data Encryption Algorithms. I: *2005 International Conference on Information and Communication Technologies*. Karachi, Pakistan 27-28 Augusti 2005, s. 84–89. DOI: 10.1109/ICICT.2005.1598556.

National Institute of Standards and Technology (NIST) (2017). *Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher*. DOI: 10.6028/NIST.SP.800-67r2.

National Institute of Standards and Technology (NIST) (2024). *Cryptographic Standards and Guidelines*. URL: https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/archived-crypto-projects/aes-development (hämtad 2025-02-25).

National Institute of Standards and Technology (NIST), Dworkin, M. J., Turan, M. S. & Mouha, N. (2023). *Advanced Encryption Standard (AES)*. DOI: 10.6028/NIST.FIPS.197-upd1.

Panda, M. (2016). Performance analysis of encryption algorithms for security. I: *2016 International Conference on Signal Processing, Communication, Power and Embedded System (SCOPES)*. Paralakhemundi, Indien 3-5 October 2016, s. 278–284. DOI: 10.1109/SCOPES.2016.7955835.

Panda, M. & Nag, A. (2015). Plain Text Encryption Using AES, DES and SALSA20 by Java Based Bouncy Castle API on Windows and Linux. I: *2015 Second International Conference on Advances in Computing and Communication Engineering*. Dehradun, Indien 01-02 Maj 2015, s. 541–548. DOI: 10.1109/ICACCE.2015.130.

Preece, J., Rogers, Y. & Sharp, H. (2016). *Interaktionsdesign. Bortom Människa-Dator Interaktion*. Studentlitteratur.

Ramakrishna, D. & Ali Shaik, M. (2025). A Comprehensive Analysis of Cryptographic Algorithms: Evaluating Security, Efficiency, and Future Challenges. I: *IEEE Access* 13, s. 11576–11593. DOI: 10.1109/ACCESS.2024.3518533.

Ramesh, A. & Suruliandi, A. (2013). Performance analysis of encryption algorithms for Information Security. I: *2013 International Conference on Circuits, Power and Computing Technologies (ICCPCT)*. Nagercoil, Indien 20-21 Mars 2013, s. 840–844. DOI: 10.1109/ICCPCT.2013.6528957.

RISK: Global Domination (2020). PC [Spel]. SMG Studio.

Routray, S., Raipure, D., Nayak K, S. R. & Shetty, N. P. (2021). Secure Sharing of Text Based Data Using Hybrid Encryption Algorithms in a Client-Server Model. I: *2021 Fourth International Conference on Electrical, Computer and Communication Technologies (ICECCT)*. Erode, Indien 15-17 September 2021, s. 1–7. DOI: 10.1109/ICECCT52121.2021.9616671.

Salama, D., Kader, H. A. & Hadhoud, M. (2011). Studying the Effects of Most Common Encryption Algorithms. I: *International Arab Journal of e-Technology* 2(1), s. 1–10.

Sanap, S. D. & More, V. (2021). Analysis of Encryption Techniques for Secure Communication. I: *2021 International Conference on Emerging Smart Computing and Informatics (ESCI)*. Pune, Indien 5-7 March 2021, s. 290–294. DOI: 10.1109/ESCI50559.2021.9396926.

Schneier, B. (1994). Description of a new variable-length key, 64-bit block cipher (Blowfish). I: *Fast Software Encryption*. Cambridge, Storbritannien 9-11 December 1993, s. 191–204. DOI: 10.1007/3-540-58108-1_24.

Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C. & Ferguson, N. (1998). *Twofish: A 128-Bit Block Cipher*. URL: https://www.schneier.com/wp-content/uploads/2016/02/paper-twofish-paper.pdf (hämtad 2025-02-26).

Unity Technologies (2024a). *Unity*. Version 6000.0.30. Game development platform. URL: https://unity.com/.

Unity Technologies (2024b). *Unity Relay*. Version 1.0.5. URL: https://unity.com/products/relay.

Yassein, M. B., Aljawarneh, S., Qawasmeh, E., Mardini, W. & Khamayseh, Y. (2017). Comprehensive study of symmetric key and asymmetric key encryption algorithms. I: *2017*

International Conference on Engineering and Technology (ICET). Antalya, Turkiet 21-23 Augusti 2017, s. 1–7. DOI: 10.1109/ICEngTechnol.2017.8308215.

Appendix A – Slumpmässiga truppattacker

```
private List<ArmyOrder> CreateRandomOrders(int size)
{
    List<ArmyOrder> orders = new();
    NodeSO startNode = map.Nodes[Random.Range(0, map.Nodes.Count)];

    for (int i = 0; i < size; i++)
    {
        orders.Add(new ArmyOrder
        {
            AttackCount = Random.Range(1, 100),
            OriginTerritory = startNode.ID,
            TargetTerritory = startNode.Connections[Random.Range(0, startNode.
                Connections.Count)].ID,
            PlayerId = 0
        });
    }

    return orders;
}
```

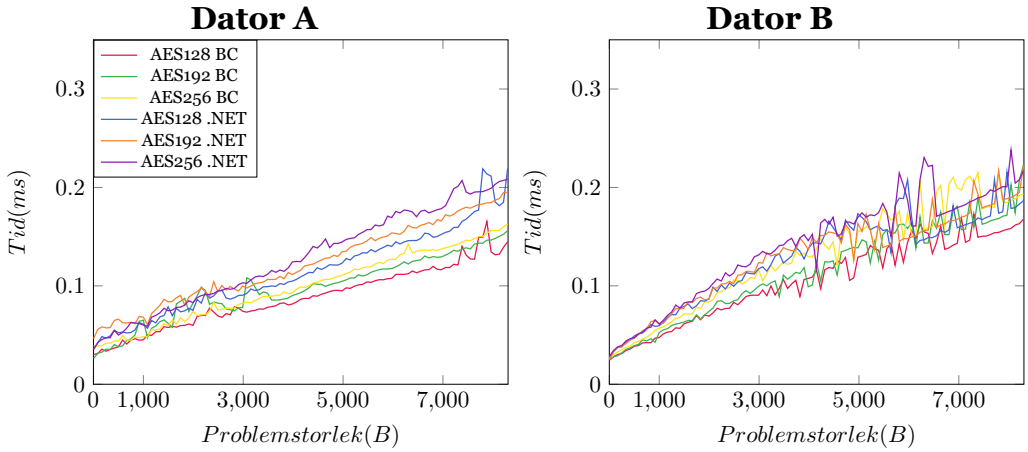
Appendix B – .NET-implementation

```
public byte[] Encrypt(List<ArmyOrder> orders)
{
    ICryptoTransform encryptor = _algorithm.CreateEncryptor();

    using MemoryStream msEncrypt = new();
    using (CryptoStream csEncrypt = new(msEncrypt, encryptor, CryptoStreamMode.
        Write))
    {
        using (StreamWriter swEncrypt = new StreamWriter(csEncrypt))
        {
            swEncrypt.WriteLine(orders.Count);
            foreach (ArmyOrder armyOrder in orders)
            {
                swEncrypt.WriteLine(armyOrder.AttackCount);
                swEncrypt.WriteLine(armyOrder.OriginTerritory);
                swEncrypt.WriteLine(armyOrder.TargetTerritory);
                swEncrypt.WriteLine(armyOrder.PlayerId);
            }
        }
    }
    return msEncrypt.ToArray();
}

public List<ArmyOrder> Decrypt(byte[] data)
{
    ICryptoTransform decryptor = _algorithm.CreateDecryptor();
    List<ArmyOrder> orders = new();

    using MemoryStream msDecrypt = new(data);
    using CryptoStream csDecrypt = new(msDecrypt, decryptor, CryptoStreamMode.
        Read);
    using StreamReader srDecrypt = new(csDecrypt);

    int orderCount = int.Parse(srDecrypt.ReadLine());
    for (int i = 0; i < orderCount; i++)
    {
        int attackCount = int.Parse(srDecrypt.ReadLine());
        string origin = srDecrypt.ReadLine();
        string target = srDecrypt.ReadLine();
        int playerId = int.Parse(srDecrypt.ReadLine());

        orders.Add(new ArmyOrder
        {
            AttackCount = attackCount,
            OriginTerritory = origin,
            TargetTerritory = target,
            PlayerId = playerId
        });
    }

    return orders;
}
```

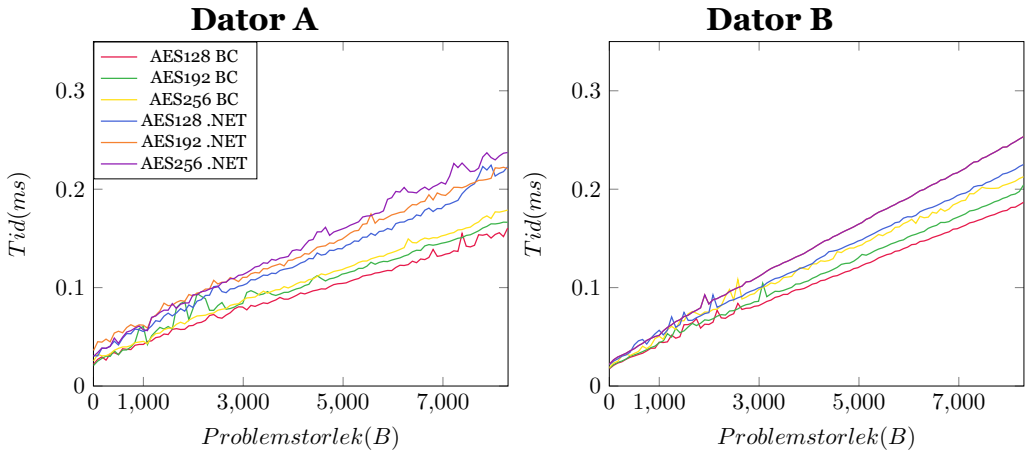
Appendix C – Bouncy Castle-implementation

```
public byte[] Encrypt(List<ArmyOrder> orders)
{
    byte[] data = ArmyOrder.ToByteArray(orders);

    IBlockCipherMode symmetricBlockMode = new EcbBlockCipher(_engine);
    IBlockCipherPadding padding = new Pkcs7Padding();

    PaddedBufferedBlockCipher ecbCipher = new PaddedBufferedBlockCipher(
        symmetricBlockMode, padding);

    ecbCipher.Init(true, _parameters);
    int blockSize = ecbCipher.GetBlockSize();
    byte[] cipherTextData = new byte[ecbCipher.GetOutputSize(data.Length)];
    int processLength = ecbCipher.ProcessBytes(data, 0, data.Length,
        cipherTextData, 0);
    int finalLength = ecbCipher.DoFinal(cipherTextData, processLength);
    byte[] finalCipherTextData = new byte[cipherTextData.Length - (blockSize -
        finalLength)];

    return cipherTextData;
}

public List<ArmyOrder> Decrypt(byte[] data)
{
    IBlockCipherMode symmetricBlockMode = new EcbBlockCipher(_engine);
    IBlockCipherPadding padding = new Pkcs7Padding();

    PaddedBufferedBlockCipher ecbCipher = new PaddedBufferedBlockCipher(
        symmetricBlockMode, padding);

    ecbCipher.Init(false, _parameters);
    int blockSize = ecbCipher.GetBlockSize();
    byte[] plainTextData = new byte[ecbCipher.GetOutputSize(data.Length)];
    int processLength = ecbCipher.ProcessBytes(data, 0, data.Length,
        plainTextData, 0);
    int finalLength = ecbCipher.DoFinal(plainTextData, processLength);
    byte[] finalPlainTextData = new byte[plainTextData.Length - (blockSize -
        finalLength)];

    return ArmyOrder.FromByteArray(plainTextData);
}
```

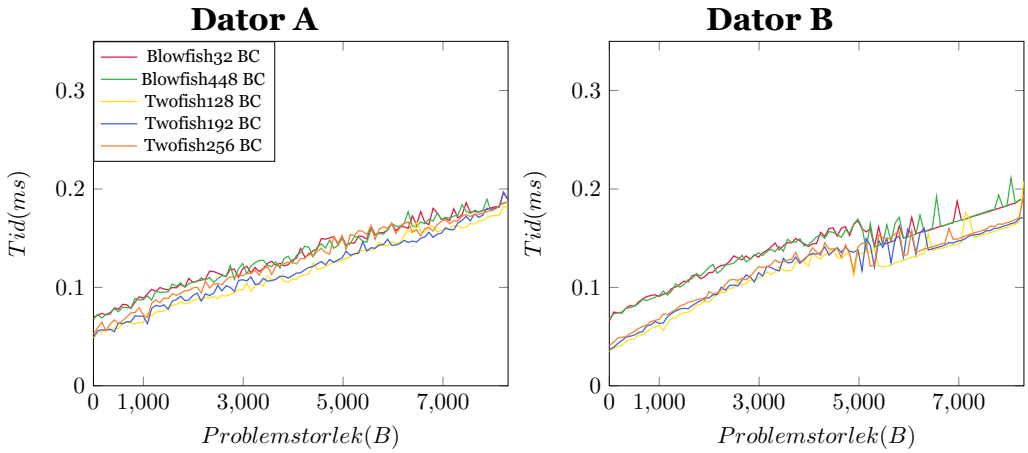
Appendix D – Exekveringstid



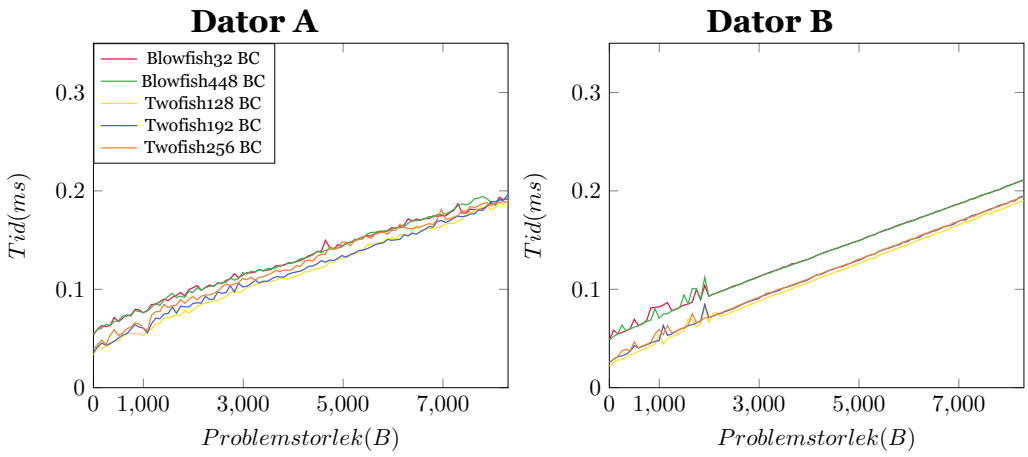
Figur 13: Krypteringstid för AES



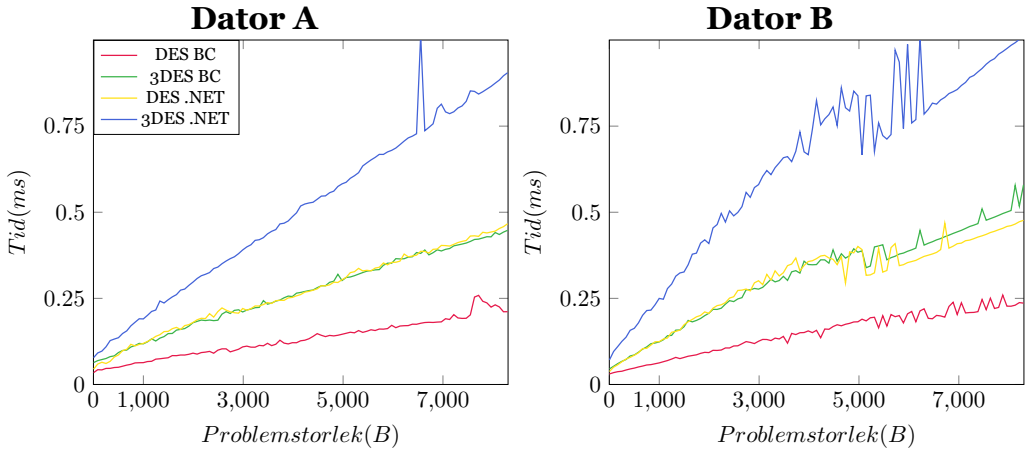
Figur 14: Dekrypteringstid för AES



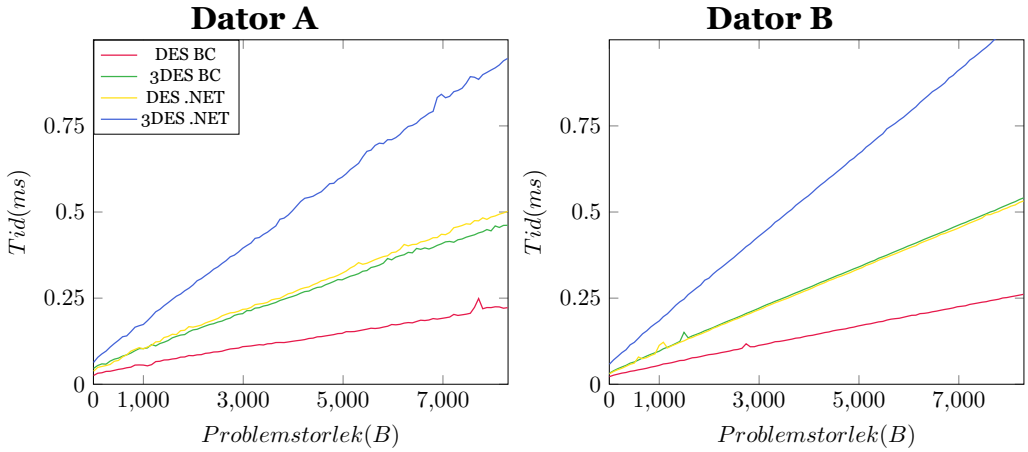
Figur 15: Krypteringstid för Blowfish och Twofish



Figur 16: Dekrypteringstid för Blowfish och Twofish

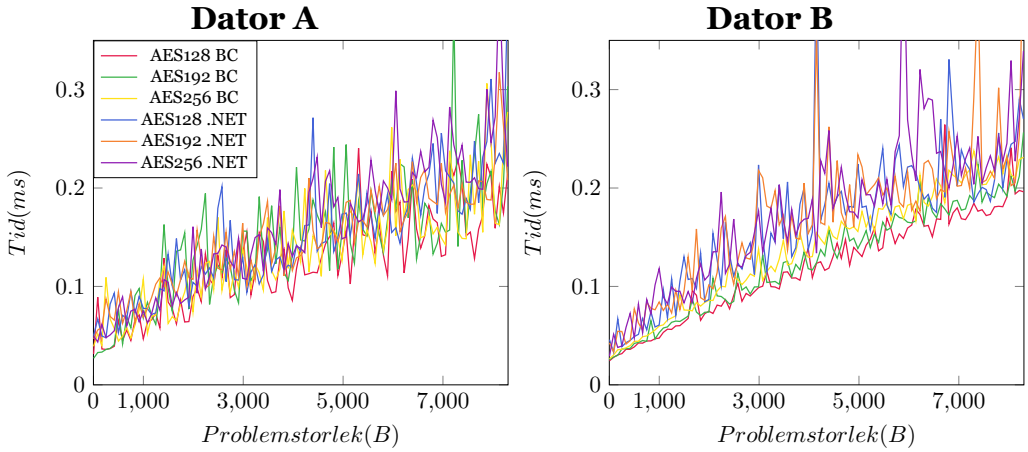


Figur 17: Krypteringstid för DES och 3DES

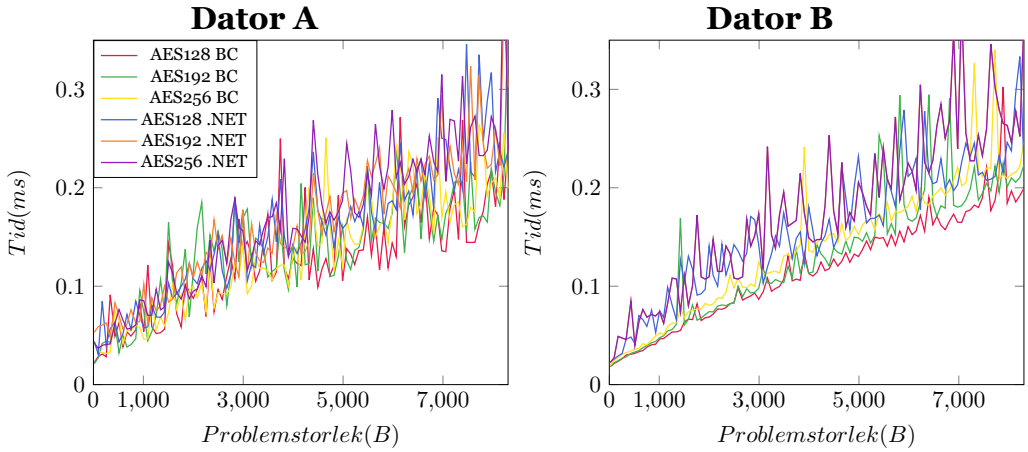


Figur 18: Dekrypteringstid för DES och 3DES

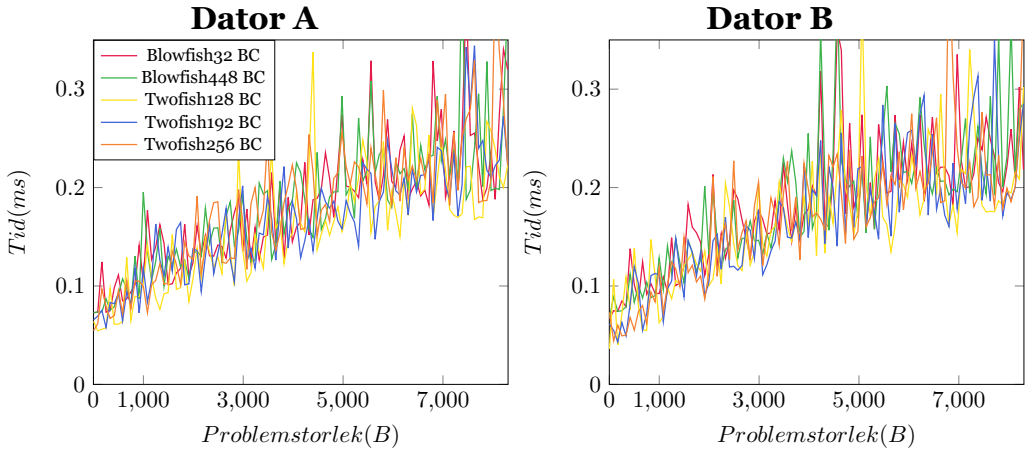
Appendix E – Exekveringstid med extremvärden



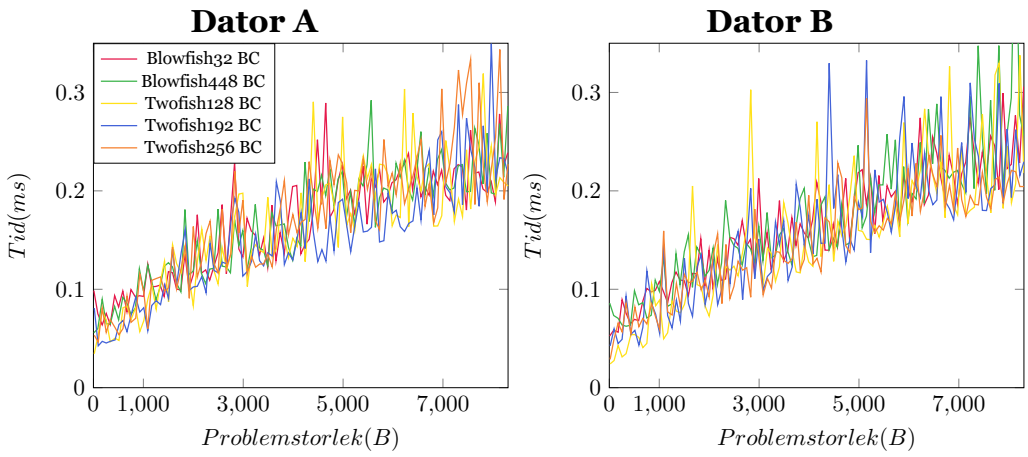
Figur 19: Krypteringstid för AES med extremvärden



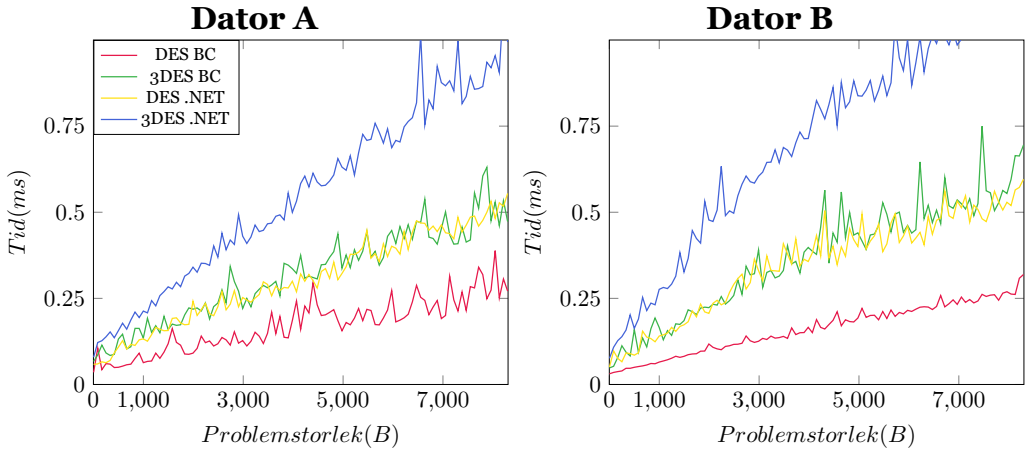
Figur 20: Dekrypteringstid för AES med extremvärden



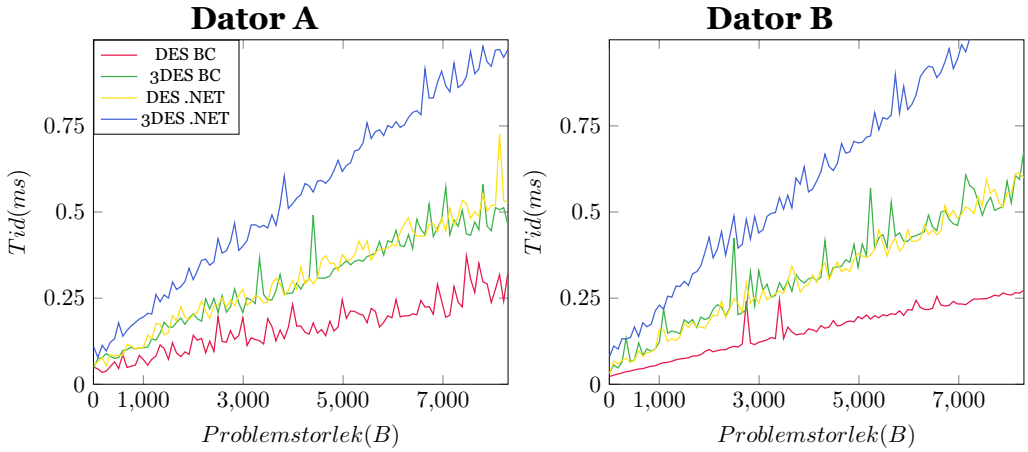
Figur 21: Krypteringstid för Blowfish och Twofish med extremvärden



Figur 22: Dekrypteringstid för Blowfish och Twofish med extremvärden



Figur 23: Krypteringstid för DES och 3DES med extremvärden



Figur 24: Dekrypteringstid för DES och 3DES med extremvärden