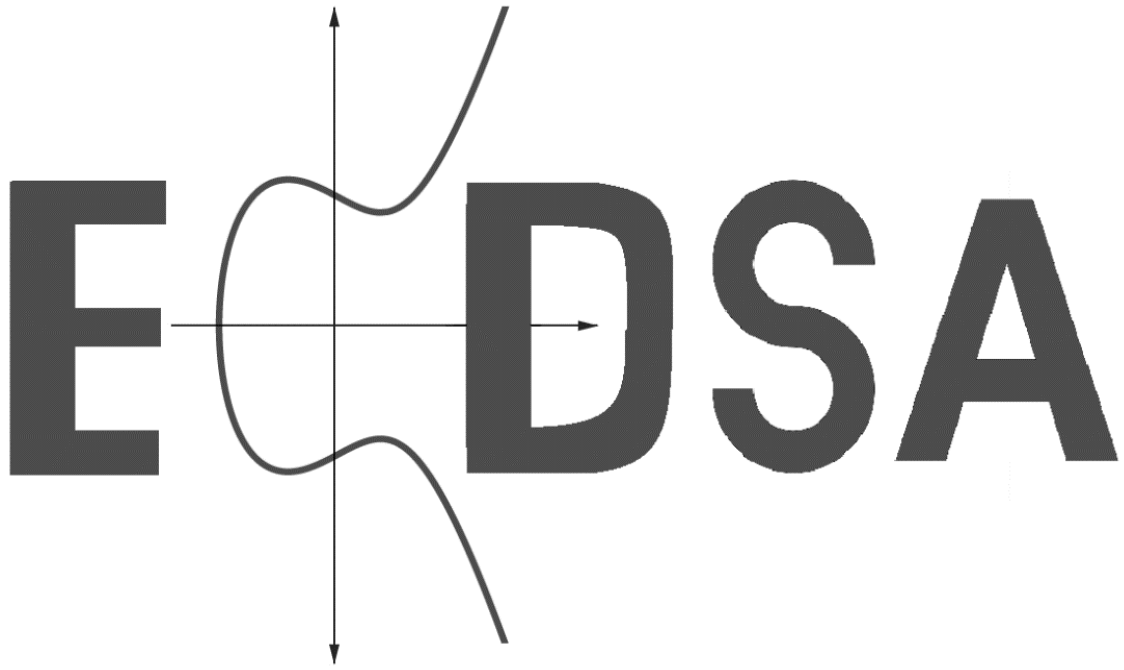
Elliptic Curve kryptering i Bitcoin



Resumé
Dette projekt omhandler Bitcoin og Bitcoins teknologi. Her undersøges hvordan kryptering, Blockchain og hashing-funktioner spiller sammen med Bitcoins teknologi. Igennem projektet gennemgår vi de forskellige protokoller, algoritmer og parameter som bruges i teknologierne i Bitcoin. Undervejs besvares spørgsmål som hvordan fungerer Bitcoin? Hvad gør det anderledes? Hvor sikker er Bitcoin? Hvem ejer Bitcoins og hvordan er de reguleret? Når Bitcoins sikkerhed bliver analyseret, vurderes sikkerheden i diskussionsafsnittet, hvor introduktionen til kendte angreb bliver præsenteret. Disse angreb inkluderer *birthday attacks, 51% attack* samt en hurtig redegørelse for Elliptic-Curve Discrete Logarith Problem (ECDLP).

Efter redegørelsen af diverse teoretiske metoder, bliver det muligt at kunne implementere algoritmerne i et praktisk eksempel, konstrueret i Typescript med fokus på Elliptic-Curve Digital Signature algorithm (ECDSA). Vi opsatte også krypteringsmetoden RSA op imod Elliptic curve kryptering (ECC), samt kunne konkludere at nøglelængderne mellem RSA og ECC var den drivene faktor for ECC's bedre ydeevne.

Indholdsfortegnelse

1. Indledning

Bitcoins popularitet og andre kryptovalutaer, har fået rampelyset det sidste årti. Bitcoinoptimister argumenterer for at bitcoin vil fuldkommen ændre verdens betalingssystem, økonomi og politik verden over. Hvor andre mener at Bitcoin vil kollapse og bare være en døgnflue. Men før vi kan tage stilling til nogen af disse argumenter, bliver vi nødt til at forstå hvordan Bitcoin fungerer.
Ved gennemgang af hvordan Bitcoin fungerer på et teknisk niveau, prøver vi at svare på nogle vigtige spørgsmål omkring Bitcoin. Hvordan fungerer Bitcoin? Hvad gør det anderledes? Hvor sikker er Bitcoin? Hvem ejer Bitcoins og hvordan er de reguleret?
For at rigtigt at kunne forstå hvordan Bitcoin fungerer, introduceres forskellige matematiske og datalogiske metoder. Disse forskellige metoder vil have relevans for at forklare teknologien bag Bitcoin, samt svare på nogle af de stillet spørgsmål. Når vi prøver at svare på nogle af vores stillet spørgsmål, sammenlignes Bitcoins teknologi mod andet tilgængeligt teknologi. Dette gøres for at se om teknologien bag Bitcoins, virkelig er så banebrydende, i forhold til andre teknologier.

Opgaveformulering

Hvordan kan man bruge eliptic curve digital signaturealgorithm (ECDSA) til at skabe sikkerhed i blockchainteknologi?

- Gør rede for kryptering og blockchainteknologi, herunder nøglefordeling.
- Redegør for matematikken bag elipticcurves og finitefields. Kom herunder ind på Secp256k1.
- Implementer en løsning i et udvalgt programmeringsmiljø, som kan kryptere og dekryptere vha. Secp256k1. Test og vurder løsningen op imod en eksisterende krypteringsimplementation f.eks. RSA.
- Diskuter sikkerheden i blockchainteknolgien og sammenlign med en eller flere udvalgtekrypteringsmetoder som f.eks. RSA.

2. Metode

Gennem denne opgave har jeg brugt diverse metoder til både programmering af mit projekt i praksis, samt matematiske metoder til teorien bag Bitcoin.

2.1. Objektorienteret programmering & Pseudokode

Jeg har brugt Objektorienteret (OOP) til at opdele mit projekt i klasser. Dette har jeg gjort for nem overførsel af diverse algoritmer og funktioner til forskellige klasser og funktioner. Klasserne har også til formål at give en overskuelig kode, samt nem implementering af klasserne.
Ved implementering af forskellige algoritmer, bruges pseudokode til at fremvise en overskuelig gennemgang af algoritmen.

2.2. Test driven Development

For at kunne garantere rigtig implementering af algoritmer til kode, har jeg brugt Test-driven Development (TDD). Formålet med at anvende TDD, er at sikre at mine funktioner fungerer i forhold til de teoretiske algoritmer og protokoller. Jeg har opbygget diverse ”test cases” som er blevet lavet før funktionerne, hvor funktionerne bliver testet for om de giver de rigtige værdier i forhold til teorien.

2.3. Matematiske metoder

Jeg bruger den Aksiomatisk-deduktive metode, med brug af grundlæggende begreber og definitioner. Samtidig bruger jeg notationer, symboler og afklaringer til at bedre beskrive de diverse abstrakte matematiske formler og definitioner. Nogle af disse definitioner bruger jeg til at udlede sætninger, som har til formål at udlede en påstand eller en forklaring. Til andre matematiske koncepter bruger jeg modellering for at og afgrænse konceptet, for at opnå en bedre forståelse.

3. Teknologierne bag Bitcoin

I dette afsnit bliver vi introduceret til hovedteknologierne inden for den decentraliseret digitale valuta Bitcoin. Bitcoin blev lavet i 2009, da den anonyme forfatter Satoshi Nakamoto udgav den dybdegående rapport *Bitcoin: A Peer-to-Peer Electronic Cash System*. [1]
Den amerikanske befolkning havde oplevet et kollaps af det finansielle system, som var til grund for dårlig forvaltning af banker og centrale enheder, som drev systemet. Blandt folks manglende tillid til systemet, bragte Satoshi Nakamoto et nyt system på banen. [2]
Satoshi Nakamoto kom med et nyt system som gav mulighed for et nyt betalingssystem og en fuldstændig digital valuta (også kaldt *Kryptovalutaer*). Betalingssystemet er et decentraliseret bruger-til-bruger betalingssystem, som køres af brugerne af Bitcoin, uden nogen form for mellemmand eller central enhed. Betalingssystemet er ikke eget af nogen, og Bitcoin kan kun fungere med komplet konsensus mellem alle brugere. Bitcoin-netværket er en offentlig regnskabsbog, hvor alle transaktioner bliver gemt. Hver bruger har sin egen Bitcoin pung samt en Bitcoin-adresse som er tilknyttet til pungen. Bitcoin-adressen gør det muligt for brugeren at modtage bitcoins fra andre via denne adresse. Autenticiteten for hver transaktion beskyttes med digitale signaturer, som kun ejeren af Bitcoin pungen har kontrol over med deres egen ”nøgle”. [3]
Hovedteknologierne som Bitcoins skellettet er blivet opbygget af, er både Kryptering, Hashing og Blockchain-teknologi. Alle tre emner kommer vi til at høre mere om, samt hvorfor disse har relevans for at opnå alle de sikkerhedsforanstaltninger der skal til at sikre transaktioner i Bitcoin.

Vi bliver også introduceret til tre fiktive personer, Alice, Bob og Eve. Som i mange andre bøger om det samme emne, bliver det antaget at Alice og Bob gerne vil dele en eller anden form for information mellem hinanden. Hvor Eve ønsker at aflytte eller manipulere med informationen mellem Alice og Bob. De fiktive personer antages nødvendigvis ikke for at være rigtige mennesker, men kan også være computere over et netværk. Moderne kryptering som bliver brugt i dag, er fokuseret på flere forskellige problemer, men de vigtigste er:

1. **Fortrolighed:** En besked mellem Alice og Bob kan ikke blive læst af andre.
2. **Autenticitet:** Bob er sikker på at Alice har sendt en besked, som Bob har modtaget.
3. **Integritet:** Bob er sikker på at beskeden fra Alice ikke er blevet manipuleret med.
4. **Ingen-benægtelse:** Det er ikke muligt for Alice at fortryde, og sige at hun ikke har sendt en besked.

Vi har mulighed for at se hvorfor disse fire punkter er vigtige, hvis vi opstiller følgende scenarie. Alice ønsker at oprette en købsordre fra Bob over internettet. Alice sender en købsordre til Bob, hvor Alices kontooplysninger er vedhæftet. Alice kræver at kommunikationen mellem Alice og Bob er hemmeligt, da Alice ikke vil have nogen andre til at vide hendes kontooplysninger eller hvad hun køber. Samtidig skal Bob være sikker på at det er Alice som har foretaget købet, og ikke en bedrager. Bob skal også kunne være sikker på at Alice ikke kan fortryde og sige hun ikke har sendt købsordren. Både Alice og Bob skal også være sikker på at integriteten af beskederne og købsordren mellem hinanden holder stik. F.eks. må købsbeløbet mellem Alice og Bob ikke manipuleres med, af en tredje part. [4]

3.1. Kryptering

Kryptering er en teknologi, der bruges til at sikre data og kommunikation ved hjælp af kryptografiske algoritmer. Det er en grundlæggende teknologi inden for Bitcoin, da kryptering bruges til at sikre, at data gemt i en Bitcoins blockchain (Se afsnit 3.3) er sikker og beskyttet mod uautoriseret adgang eller ændring i blockchainen.
Krypteringsmetoden har mulighed for kun at give autoriserede personer adgang til data, som kun dem med den korrekte nøgle kan dekryptere til den originale data. På denne måde beskytter kryptering data mod uautoriseret adgang fra hackere, uautoriserede personer eller andre former for spionage.

Nøglefordeling er en central del af kryptering, for at skabe sikkerhed for at sende krypteret data over internettet. [5] Der er to former for nøglefordeling i kryptering, symmetrisk og asymmetrisk nøglekryptering. En symmetrisk nøgle er den ældste form for krypteringssystem, som blev brugt i blandt andet Enigma maskinen i Anden Verdenskrig. [6]
Symmetrisk kryptering bruger samme nøgle til både at kryptere og dekryptere data. Dette betyder derfor at både sender og modtager af et stykke data deler samme nøgle, eller en simpel transformation af nøglen. Symmetrisk nøglekryptering har generelt en mindre nøglestørrelse, som gør metoden hurtig og effektiv, på grund af den mindre lagerplads brugt på nøglen til hurtigere transmission. Men dette er også ulempen ved symmetrisk nøglekryptering, alle autoriseret parter skal kende nøglen før metoden kan fungere. Dette kan være problematisk, fordi nøglen skal sendes på en sikker måde, så den ikke kan kompromitteres og spioneret af en tredjepart. [7]

Asymmetrisk nøglekryptering, også kendt som ”offentlig-nøglekryptering", er en metode hvor der bruges to nøgler: en offentlig nøgle, der er tilgængelig for alle, og en privat nøgle, der kun er tilgængelig for ejeren af nøglen. Nøglesættet bestående af en privat og en offentlig nøgle, som bliver genereret med matematiske funktioner, hvor sikkerheden af nøglerne kommer fra funktioner som er lette at beregne den ene vej, men svær den anden vej. Denne form for funktion kaldes ”One-way function”, med forskellige variationer til konceptet såsom en ”Trapdoor function”, hvor der er mulighed for at få adgang til inputtet ved brug af en privatnøgle. [8]

3.2. RSA-Kryptering

Den mest kendte asymmetriske krypteringsalgoritme som også er en ”Trapdoor function” er RSA, udviklet af *Rivest Shamir* og *Adleman* i 1977. [9]

Metoden for at generere et nøglesæt i RSA, fungerer ved at vælge to store primtal, p og q og derefter beregn deres produkt:

$$n = p \cdot q$$

Derefter udregnes Euler’s totient funktion $\varphi(n)$:

$$\varphi(n) = (p-1) \cdot (q-1)$$

$\varphi(n)$ finder antallet af ”coprimes” til n. Coprimes er to tal som ikke har nogen fælles faktorer, tal som er Indbyrdes primisk [10] [11]

Derefter vælges et tal e, der er primtalsfaktor med $\varphi(n)$ ved brug af Euler’s totient funktion:

$$e = 0 < e < \varphi(n) \text{ og } gcd\big(e, \varphi(n)\big) = 1$$

gcd (Greatest common divisor) er det største tal som begge tal går op i (dermed den største divisor til e og $\varphi(n)$) [12]

Tallet e kaldes for den offentlige exponent, da e er en del af den offentlige nøgle. Euler's totient funktion bruges kun til at beregne den entydige løsning d:

$$(d \cdot e) \equiv 1 \ mod \ \varphi(n)$$

Symbolet ($\equiv$) beskriver modulær kongruens, hvor $(d \cdot e) \ mod \ \varphi(n)$ og $1 \ mod \ \varphi(n)$ har samme rest. Dette kaldes også for en *Abelian gruppe* (se definition 2. - afsnit 4.1).

Dette tal d kaldes den private exponent. Den offentlige nøgle er så (n, e) og den private nøgle er (n, d). [9]

Hvis Alice først vil sende en besked m til Bob, skal hun kryptere besked til den krypteret besked c ved:

$$c \equiv m^e \ (mod \ n)$$

Bob kan derefter dekryptere Alices originale besked ved:

$$c^d \equiv (m^e)^d \equiv m \ (mod \ n)$$

Bob har nu dekrypteret Alices besked m.

RSA bliver ikke brugt som kryptering til nøglegenerering i Bitcoins tilfælde, da RSA har flere ulemper som vi diskuterer senere. (Se afsnit 6.) Derimod bruger Bitcoin *Elliptic-Curve Digital signature algorithm* (ECDSA), som bruger matematiske funktioner og operationer til generering af offentlige nøgler og signaturer. (Se afsnit 4.5)

3.2.1. Hashing

Hashing er en vigtig del af kryptering og sikre at data og beskeder forbliver sikkert og ulæseligt for uautoriseret bruger, men også at beskeden kommer fra den rigtige bruger. Hashing er en algoritme som beregner en fast størrelse bits som en tekst. Denne tekst bliver beregnet ud fra en besked, fil eller anden form for data. Hashværdien er en form for fortyndet resumé af alt givende data.
En vellykket hashing algoritme ville have egenskaben til at drastisk kunne ændre hashværdien, selv hvis kun en enkel bit eller byte i givende data bliver ændret. Hvis ikke dette lykkes, ses det som værende en dårlig hashing-algoritme, med dårlig tilfældighed som er let at kunne knække. En vellykket hash-algoritme bør være kompleks nok, til der ikke bliver produceret den samme hashværdi fra samme datastykke. Hvis den samme hashværdi forekommer efter det fuldkommen samme stykke data, kaldes det en hash-kollision. Før en hash-kollision bør forekomme, skal chancen være minimal, før algoritmen kan betragtes som sikker.

Den producerede hashværdi fra hashing-algoritmen, bliver ofte fremvist som en hexadecimal streng med flere tegn. Hashing metoden er samtidig en One-way function, så den produceret data kan ikke arbejde baglæns og få den originale data tilbage. Med den hexadecimale streng er det muligt at sammenligne to stykker data for ændringer eller variationer, uden selv at gennemgå dataet. Samt kan hashing bruges til verificering af datastykkets integritet, efter datastykket er blevet overført fra et sted til et andet. [13]

Der er flere forskellige typer af hash-algoritmer. En af de mest kendte hashing algoritmer, er SHA-2-familien består af seks hash-funktioner. Hver funktion I SHA-2 er med forskellige bit størrelser som hashværdi. SHA-2-familien blev udviklet i Amerika af National Security Agency (NSA) og udgivet i 2001. [14]

Hash-funktioner har flere forskellige egenskaber som gør det muligt for funktionerne at verificere integriteten af digitale signaturer i et asymmetrisk nøglesystem, såsom RSA. Den private nøgle beregnet fra RSA, $k_{priv} = (n, d)$ bruges til at underskrive beskeden M og producere en signatur S med funktion $Sign(M, k_{priv})$. Den offentlige nøgle $k_{pub} = (n, e)$ kan nu verificere signaturen S i forhold til beskeden M, med funktion $Verify(M, S, k_{pub})$ som returnere et boolsk variable om signaturen er sand eller falsk.

Selvom det kræver meget computerkraft at forfalske en signatur, kan det teoretisk godt være muligt (Se afsnit 6.3). Det er muligt at skelne mellem to former for forfalskninger, E*ksistentiel* og *Universel* forfalskning.
Et succesfuldt universelt angreb beregnes en valid signatur S ud fra beskeden M med privatnøglen k_{priv}. Dette vil betyde at angriberen kan konstruere enhver signatur ud fra enhver besked, som vil betyde at angriberen har med succes fundet den tilsvarende private nøgle fra signaturen. Angriberen har dermed forbigået krypteringen, og så vil enhver hashing algoritme ikke gavne noget. [15]
Ved at udføre et eksistentiel angreb, beregnes et validt par af M, S med den offenlige nøgle k_{pub}. For en tilfældig signatur S, er det muligt at beregne en besked M, ud fra den offentlige nøgle K_{pub} af en valid signatur:

$$M = S^e \ (mod\ n)$$

Udregning af besked M, som ville passe signaturen S, med offentlige nøgle e

Et eksempel kunne være hvis Alice underskriver en besked M_{alice} og producerer en valid signatur S. Eve vil gerne angribe Alice og kapre Alices signatur, Eve kender kun Alices offentlige nøgle (n, e) og signatur S. Eve udregner en tilfældig besked M_{eve}, med formlen $M_{bob} = S_{alice}^e \ (mod\ n)$, hvor M_{bob} vil være i stand til at producere en valid signatur med Alice offentlige nøgle.

$$S_{eve} = S_{alice}$$

Eve har produceret sin egen forfalsket signatur S_{eve} som er tilsvarende Alices signatur S_{alice}

Eve har med succes beregnet en valid signatur, med brug af Alices offentlige nøgle. Men Eves besked M_{eve} er tilfældig og er ikke det samme som Alices originale besked M_{alice}:

$$M_{eve} \neq M_{alice}$$

Eves besked er ikke det samme som Alices besked

Hashing algoritmer kan blive brugt til at sikre sig for eksistentielle angreb. Det er ikke muligt at udføre et eksistentielt angreb med brug a hashing, da som nævnt før vil en hash funktion H, altid fordøje den tilsvarende faste streng D ud fra den originale data. Derfor skal Eve besked M_{eve} opfylde Alices fordøjet besked D_{alice} ud fra Alices originale besked M_{alice}: [14]

$$H(M_{alice}) = D_{alice}$$

$$H(M_{eve}) \neq D_{alice}$$

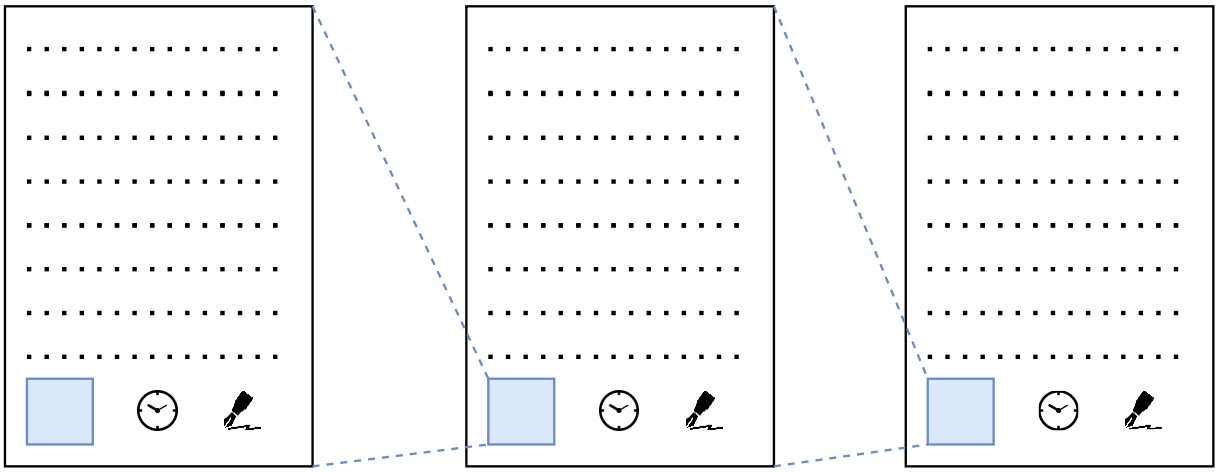
3.3. Blockchain-teknologi

Teknologien bag blockchain har en vigtig rolle i forbindelse, med at skabe den universelle regnskabsbog hvor alle transaktioner bliver sikkert gemt i Bitcoin netværket. Ideen for blockchain kan spores tilbage til 1991, med Haber og Stornettas forslag til en metode for sikker tidsstempling af digitale dokumenter. [16] Målet for metoden var at give en idé om, hvornår et dokument opstod, men vigtigst hvornår dokumentet opstod og hvilken rækkefølge. Hvis et dokument opstod før det andet, ville dette have en effekt på hele tidsstemplingen og dermed hele blockchainen. Blockchainteknologien var generelt ikke brugt til noget signifikant, før den anonyme forfatter Satoshi Nakamoto udgav Bitcoin.

Haber og Stornettas metode gav mulighed for bruger at sende dokumenter til et netværk, hvor netværket derefter vil gemme tidspunktet for modtagelse og underskrive det i det næste dokument i blockchainen. Det næste dokument vil nu indeholde en reference (ofte en hashværdi) til dataet fra det forrige dokument. Som nævnt i Hashing, vil enhver ændring i det gamle dokument, ændre hashværdien til noget nyt. Dette vil

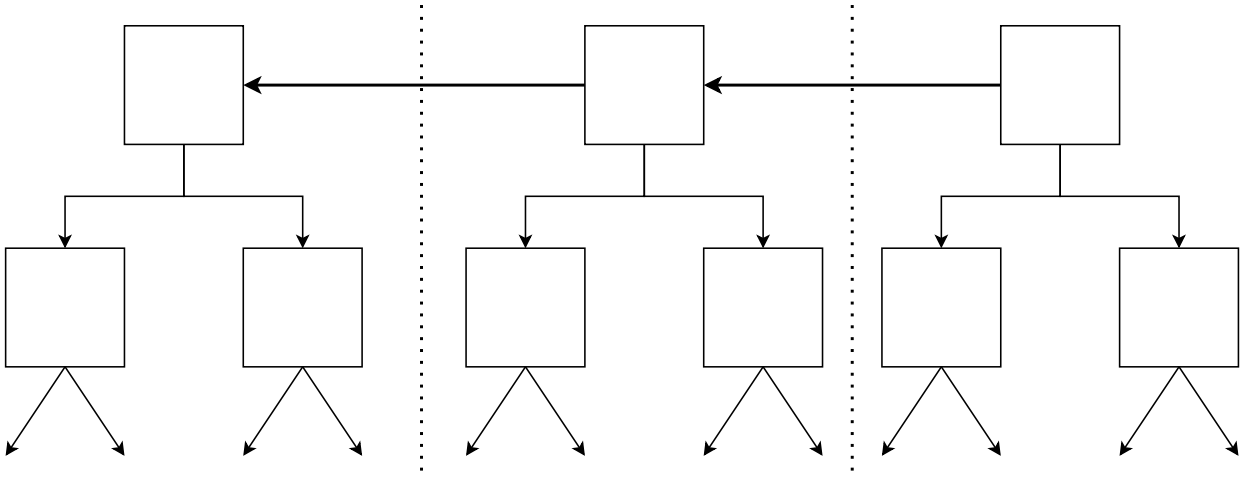
dermed også betyde at referencen i det nye dokument vil være ugyldig, fordi der er sket ændringer i det gamle dokument.

Referencen sikrer integriteten af indholdet til det forrige dokument. Referencerne kan gøres rekursivt, hver reference skaber en form for ”integritets-kæde” for alle forrige dokumenters data og historie.



Figur 1 Lineær struktur af rekursiv referencer af dokumenter. Inkludere tidspunkt for modtagelse af dokumentet af forrige dokument, nuværende tid, hashværdi til forrige dokuments og underskriver alle tre ting sammen [16]

Senere blev den lineære struktur udvidet for mere effektivitet; i stedet for at referere et dokument ad gangen, kan vi samle dem i ”Blocks”. I hver block vil en liste af en eller flere dokumenter blive samlet sammen og forbundet. Men i stedet for en lineær struktur, så blev det i form af et ”data-tree” struktur, hvor det kun er hoved-blocken som refererer til forrige hoved-block. Den udvidet metode vil formindske tiden og energien brugt på at verificere enkle dokumenter, som fremstår på et specifikt tidspunkt i blockchains historie.



Figur 2 Data-tree struktur af rekursiv referencer af blocks. Pile repræsenter referencer, mens punkteret linjer er tidsintervaller [16]

Bitcoins blockchain er opbygget af data-tree strukturen (se Figur 2), hvor blockchainen er opbygget således at hver block som skal tilføjes til blockchainen, bliver forsinket med ~10 min. Denne mindre modifikation gør det muligt at undgå kravet på store mængde af computerkraft, til verificering af nye transaktioner. Men i stedet bliver hver transaktion gemt og verificeret af individer kaldt "miners". Alle miners er en del af et stort decentraliseret netværk, hvor hver miner holder styr på blocks i blockchainen. Alle brugere på netværket kan blive en miner til at verificere blocks, ved at løse svære hashing-puslespil med computer kraft. [16] Minernes får som belønning en eller flere Bitcoins, hvis de løser opgaven. Miners er de eneste som kan skabe nye Bitcoins som kommer i cirkulation. [17] Dette skaber et transparant og decentraliseret netværk, fri for mellemmænd eller intern manipulation.

Bitcoin kombinerer i bund og grund ideen om at bruge computerkraft til at oprette nye blocks med nye transaktioner, som registreres i en offentlig regnskabsbog, som forhindrer dobbelt forbrug af Bitcoins.

4. Secp256k1 - Bitcoins krypterings protokol

Bitcoins krypterings protokol Secp256k1, er parametrene for Bitcoins Elliptic Curve nøglekryptering og Finite field. Secp256k1 var næsten aldrig brugt før Bitcoins implementering af protokollen. Protokollen blev med omtanke valgt som algoritme til Bitcoin, da protokollens parameter giver adgang til hurtig og effektive beregninger. Som resultat gav Secp256k1 mere end 30% hurtigere beregninger i forhold til andre Elliptic Curves, samt er parametrene forudsigelige som reduceret muligheden for "Backdoors" (mulighed for hemmelig adgang) af skaberen. [18]

4.1. Finite fields

Når aritmetik bliver lavet inden for Elliptic curves, er det vigtigt at implementere et Finite field (også kaldt "Galois field" eller "begrænset felt"). Et Finite field er en matematisk struktur, som indeholder en begrænset mængde af elementer. Den totale mængde af elementer i et finite field kaldes for feltets *orden* (se afsnit 4.1.1). Normalt er elementerne repræsenteret som heltal $\mathbb{Z}$ i intervallet $\{0, 1, \dots, p-1\}$, hvor p er et "stort" primtal, samt er elementerne regnet med modul p. [4]

Definition. (Field). Et Field $\mathbb{F}$ (eller "felt"), er en *ring* (se definition 4.) så for hvert element a som ikke er 0 i feltet $a \in \mathbb{F}$, findes der et andet element $b \in \mathbb{F}$ så $ab = 1$. Feks, hvis p er et primtal, så $\mathbb{F}_p$ eller $\mathbb{Z}/p\mathbb{Z}$ er et felt. [19]

$$\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$$

Finite fields konstruktion af heltal $\mathbb{Z}$ modulo p

Sættet af elementer i et Finite field overholder nogle regler (kendt som *feltaksiomer*), som er nogle matematiske grundsætninger som antages at være sande. Disse feltaksiomer i et Finite field er de to binære operationer multiplikation og addition. Hvor en additionsoperation skrives som $a + b$, og en multiplikationsoperation skrives som $a \times b$. [19]
De to feltaksiomer, addition og multiplikation skal opfylde følgende egenskaber:

Definition 1. (Gruppe). En gruppe er et set af elementer $\mathbb{F}$ med operationen $\times$ og $+$ (angivet ved addition og multiplikation nedenunder)

1. For alle $a, b, c \in \mathbb{F}$, har vi
   - $(a \times b) \times c = a \times (b \times c)$
   - $(a + b) + c = a + (b + c)$
2. For hver $a \in \mathbb{F}$, vi har $1a = a1 = a$, samt excistere $b \in \mathbb{F}$ så $ab = 1$.

Definition 2. (Abelian gruppe) En *Abelian gruppe* er hvor $ab = ba$ for hvert $a, b \in G$.

Definition 3. (Neutral element) To neutral element finds 0 og 1 i $\mathbb{F}$, hvor $a + 0 = a$ og $a \cdot 1 = a$. Det neutrale element for 0 er repræsenteret som $\mathcal{O}$.

Definition 4. (Ring). En *ring* R er et sæt af elementer, med de binære operatører $\times$ og $+$, elementer $0, 1 \in R$ så at R er en Abelina gruppe under $+$, og for alle $a, b, c \in R$. Så at:

- $(ab)c = a(bc)$.
- $a(b + c) = ab + ac$.

4.1.1. Punkter af begrænset orden

Punktorden $\#G$ definerer antallet af punkter i en gruppe. Dette betyder at orden af et enkelt element i en gruppe, er antallet af gange elementet skal multipliceres med sig selv, ind til elementet bliver til det neutrale element af gruppen. Definitionen er følgende:

Definition 5. Et element P af en gruppe har *orden* $\#G$

$$\#G = \underbrace{P + P + \cdots + P}_{sum\ af\ \#G} = \mathcal{O},$$

Hver er $\mathcal{O}$ bliver betragtet som det neutrale element i gruppen.
Et Finite field af ordenen $\#G$ excistere kun, hvis $\#G$ er et primtal opløftet i et enkelt positivt heltal p^k (også kaldt *prime power*). Hvor p er et primtal og k er et positivt heltal. [20]

4.1.2. Montgomery reduktion

En effektiv måde at implementere modulær aritmetik for et stort primtal p, er Montgomery reduktion. Montgomery algoritmen er en metode til at reducere store modulære primtal, til et mindre mere effektivt tal. Algoritmen er effektiv da metoden formår at undgå at bruge division, derfor er Montgomery reduktion god til beregning af:

$$c = a \cdot b \ (mod\ n)$$

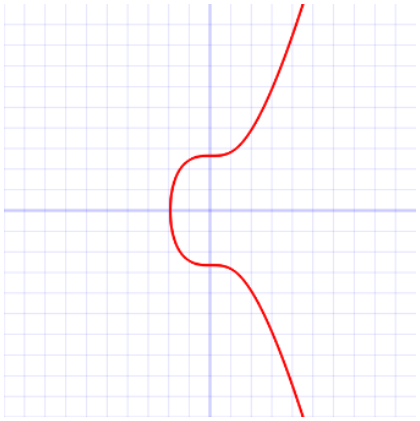
Montgomery algoritmen er nemmere håndteret af en computer på grund af at tallene efter Montgomery algoritmen er repræsenteret i binær form. For at bruge Montgomery reduktion algoritmen i et Finite field, skal vi først finde en *"Montgomery faktor" r.* Hvor faktoren r er en potens af 2 og $r > n$, hvor n er modulo. Derefter beregners inverse $r^{-1}\ (mod\ n)$, med brug af den udvidet "Euklidisk" algoritme. Når invers r er fundet, kan r derefter blive brugt til reduktion af store heltal $x\ (mod\ n)$ [21]

4.2. Elliptic Curves

En Elliptic Curves (eller elliptisk kurve) er en matematisk kurve som isomorfier (lighed med hensyn til kurves form) på x-aksen, med formen som en kort *Weierstrass*. [4]
Formlen for en elliptisk kurve K over Finite field $\mathbb{F}_p$:

$$K(\mathbb{F}_p) = \{(x, y) \in \mathbb{F}_p : y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\} \quad (1)$$

Hvor $a, b \in \mathbb{F}_p$. Her er symbolet $\mathcal{O}$ tænkt som punktet "uendelig" på kurven K. Punktet $\mathcal{O}$ bliver betragtet som det neutralt element, og som ikke er et direkte punkt på kurven K. elliptiske kurver har to operationer, *punkt addition* og *punkt duplikering,* som har til formål at skabe nye punkter på den elliptiske kurve. Punkt addition og punkt duplikering på en elliptisk kurve, kaldes for punkt aritmetik.



Secp256k1's elliptiske kurve $y^2 = x^3 + 7$ over Finite field $\mathbb{F}_p$ [18]

4.2.1. Elliptic Curve gruppestruktur & aritmetik

Før punkt aritmetik på en elliptisk kurve kan forekomme, skal vi først definere den elliptiske kurve K over feltet $\mathbb{F}_p$, hvor p er et stort primtal.

(Elliptisk kurve Gruppe lov).

Vi definerer to punkter; $P_1(x_1, y_1)$ og $P_2(x_2, y_2)$ på kurven $K(\mathbb{F}_p)$, som vi bruger til at beregne et tredje punkt R på kurven $R = P_1 + P_2 \in K(\mathbb{F}_p)$. Algoritmen til punkt addition og punkt duplikering, for at beregne det tredje punkt R, gøres følgende: [19]

1. Hvis $P_1 = \mathcal{O}$, set $R = P_2$ eller hvis $P_2 = \mathcal{O}$, set $R = P_1$
   Trin 1. viser hvis en af punkterne P er $\mathcal{O}$, sættes R lige med det andet punkt af P

2. Hvis $x_1 = x_2$ og $y_1 = -y_2$, set $R = \mathcal{O}$
   Trin 2. viser hvis $P_1 + -P_2$ sættes $R = \mathcal{O}$

3. $$\lambda = \begin{cases} \dfrac{(3x_1^2 + a)}{(2y_1)} & \text{hvis } P_1 = P_2 \\ \dfrac{(y_1 - y_2)}{(x_1 - x_2)} & \text{hvis } P_1 \neq P_2 \end{cases}$$
   Trin 3. viser hvordan vi skal beregne λ, når P_1 er lig med P_2 eller ej.
   Bemærk at hvis $P_1 \neq P_2$, er λ lig med hældningen af P_1 og P_2.

4. $$R = (\lambda^2 - x_1 - x_2, -\lambda x_3 - v)$$
   Hvor $v = y_1 - \lambda x_1$ og $x_3 = \lambda^2 - x_1 - x_2$ er x-koordinatet af R
   Trin 4. viser formlen for at få koordinaterne af punkt R, med brug af λ beregnet i Trin 3.

Vi ser i under algoritmen, at de første to trin tjekker hvorledes punkterne P er punktet uendelig $\mathcal{O}$ eller på samme lodret linje. Dette vil afgøre hvordan λ vil blive beregnet.

For at kunne få en god forståelse af algoritmen for punkt addition og duplikering, Vil vi visualisere kurven E, med punkterne P_1, P_2, P_3 og vores beregnet P_3 afspejlet af R. [4] Bemærk at det er vigtigt for punktet P_3 er en refleksion af R, da dette gør vores $+$ operation abelsk.

4.2.2. Punkt addition

Hvis $P_1 \neq P_2$ beregnet vi summen af $P_1 + P_2$, hvor λ er hældningen mellem P_1 og P_2. Vi får derefter et tredje punkt R som skær kurven K, samt mellem linjen L som er defineret fra punkterne P_1 og P_2. Det beregnet punkt R er derefter punktet reflekteret på x-aksen, som er vores endelige resultatet P_3.

$$P_3 = P_1 + P_2 \tag{2}$$

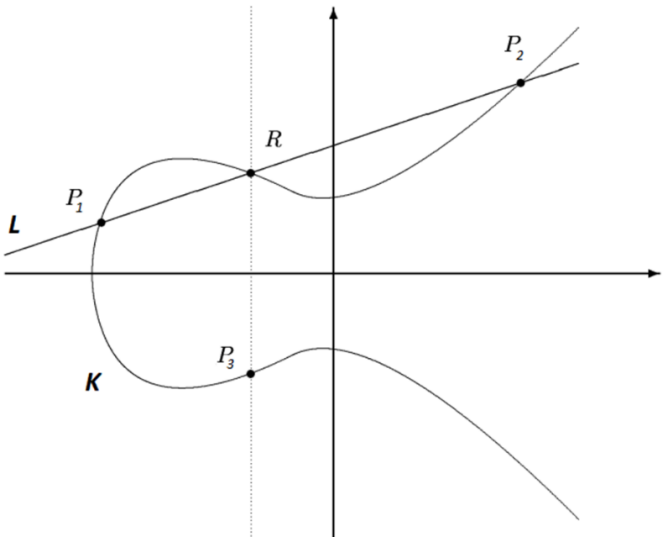


Illustration af punkt addition, mellem to punkter P_1, P_2 og kurve K [4]

4.2.3. Punkt duplikering

Hvis $P_1 = P_2$, skal vi ligge P til sig selv, eller for at *duplikere* P, tag vi tangenten til kurven K ved punktet P. Tangentlinjen til punktet P vil have et, og kun et skæringspunkt R med kurven K. Igen bliver skæringspunktet R reflekteret på x-aksen, til at beregne vores endelige resultatet $[2]P$.

$$[2]P = P + P \tag{3}$$

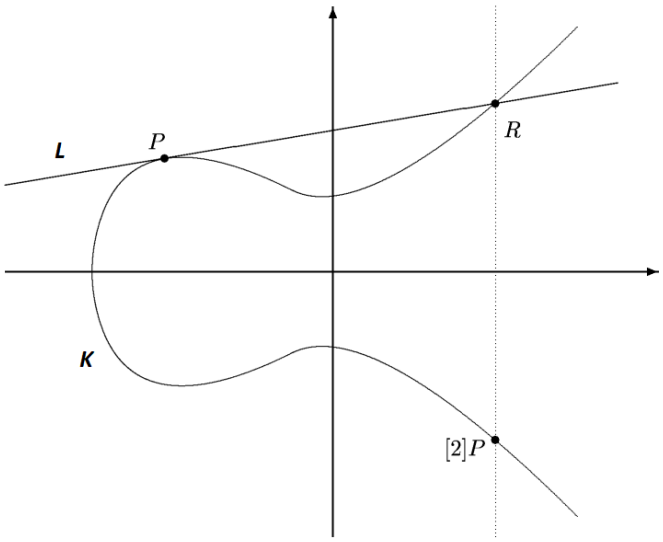


Illustration af punkt duplikering af et punkt P og kurve K [4]

4.2.4. Punkt multiplikation

Når en bruger f.eks. skal generere sin offentlige nøgle Q, skal brugeren gør brug af punktoperationerne addition og duplikering. Det vil være mest effektivt i et kryptografisk system at bruge punktoperationerne flere gange, til at øge sikkerheden for at finde løsningen på ECDLP (Se afsnit 6.1). [4]
En effektiv måde kunne udøve punktoperationerne på er med punkt multiplikation. Den mest almindelige punktmultiplikations-algoritme er *Double-and-Add*. Double-and-Add Multiplicere et punkt P på en elliptisk kurve K over finite field $\mathbb{F}_p$ med en skalar k.

$$Q = [k]P = \underbrace{P + P + \cdots + P}_{k\ gange}, \tag{4}$$

Hvor P, er punktet på kurven, k er et arbitrært heltal i intervallet $1 \leq k < p$. Men skalaren k bliver repræsenteret som binær i algoritmen, da binær giver effektivitetsfordele og lavere tidskompleksitet. [22]

Q er det resulterende punkt af P lagt sammen k gange.
En anden populær punkt multiplikations algoritme som bygger videre på Double-and-Add algoritmen, er *Montgomery Point multiplication*. Både Montgomery- og Double-and-Add algoritmen bruger samme princip i multiplikation af et punkt P med k. Men Montgomery kan være hurtigere i nogle tilfælde, da Montgomery ikke behøver at håndtere punkternes y-koordinater. [23] [24]

Montgomery Double-and-Add Algorithm

Input: Punkt $P \in E(\mathbb{F}_n)$, længden i bits af et arbitrært heltal k.
Output: Punkt Q.

1. $Q \leftarrow \mathcal{O}, R \leftarrow P$
2. for k nedtil 0:
3. hvis $k = 1$:
4. $Q = P + Q$
5. ellers:
6. $R = [2]P$

Returnerer Q

4.3. Elliptic Curves baserede protokoller

I dette kapitel forklares om elliptiske kurvers basale protokoller, med fokus på Elliptic-Curve Digital signature algorithm (eller ECDSA). ECDSA er en af elliptisk kurvers basale kryptografiske protokoller, som elliptiske kurver primært bruger. ECDSA omhandler kun to fokuspunkter; signaturer (Underskrivning og Verificering af dokument) og kryptering af nøgler.

Standardisering af kryptografiske protokoller, er vigtig for at udvide et krypteringssystem i større skala. Dette er vigtig da der er mange bruger i systemet, kan og vil have forskellige hardware og software kombinationer, samt for at skabe en sikker protokol. Når vi snakker om standardisering i elliptiske kurver, er det både hvordan hver algoritme fungerer, samt formatet af det sendt data. Formatet af dataet er vigtigt for transparenthed i det kryptografiske system.
Der findes flere forskellige standardiseringer for forskellige elliptiske kurver. Disse standardiseringer hjælper udvikler af kryptografiske systemer, samt giver det transparenthed for brugere af systemerne. Standardiseringerne er samtidig valgt af forskellige eksperter, som er blevet enige om parameter for at opnå en vis sikkerhed for protokollerne. [23]

De mest relevante standardiseringer inden for elliptiske kurver er følgende standarter:

- $\boldsymbol{IEEE\ 1363}$: Denne standart indeholder alle algoritmer til offentlige nøgler, med blandt andet fokus på ECDSA. Ud over dette bringer standarten også lys på de basale talteoretiske algoritmer (såsom primtal, moduls osv.), som bruges i kryptering af offentlige nøgler.
- $\boldsymbol{ANSI\ X9.62 + X9.63}$: Disse to standarter specificerer hvilke beskedformater som skal bruges i ECDSA, samt en liste af foreslået elliptiske kurver.
- $\boldsymbol{FIPS\ 186.2}$: Denne standard specificerer signaturer for både DSA og ECDSA, samt en liste af foreslået elliptiske kurver som bruges hos den Amerikanske regering.
- $\boldsymbol{SECG}$: SECG-standarten spejler næsten alt indhold for ANSI-standarterne. Men SECG gør det mere læsbart og det er offeligt tilgængeligt, fra hjemmesiden https://www.secg.org/
- $\boldsymbol{ISO}$: ISO-standarten indeholder en relevant standart som har relevans for ECDSA.

4.4. Digital Signature Algorithm (DSA)

ECDSA er en variant af "Digital Signature Algorithm" (DSA). Før vi går i dybden med ECDSA, ville det være illustrativt at gennemgå DSA først. [23]

For at vi kan bruge DSA-algoritmen til signaturer og definering af offentlige nøgler, skal vi først generere parametrene til vores DSA-system. Først vælges først en hash-funktion H, som har output af en streng på $|H|$ længde bits. Derefter defineres en primtal q over $|H|$ bits, samt endnu et primtal p hvor

$$p - 1\ (mod\ q) = 0 \tag{5}$$

Derefter beregnes vores "generator" g ved først at finde et tilfældigt $h \in \{2 \dots p-2\}$, også:

$$g = h^{(p-1)/q}\ (mod\ p) \tag{6}$$

Typisk bruges hashing funktionen SHA-1 [$\mathbf{FIPS\ 180.1}$], men med udgivelse af nyere hashing-funktioner, bliver SHA-256, SHA-384 og SHA-512 [$\mathbf{FIPS\ 180.2}$] også brugt til formål af give en større værdi for m.

De fire parameter (H, p, q, g) for vores system kan nu deles til store mængder af bruger i vores system. Brugerne kan nu med de fire parametre, generere sin offentlige nøgle og begynde at lave signatur beskeder over systemet.

For at lave en offelig nøgle med brug af vores system, skal der først vælges en privatnøgle d, hvor d er inde for intervallet $0 \leq d < q$. Derefter kan den offentlige nøgle Q beregnes.

$$Q = g^d\ (mod\ p) \tag{7}$$

Det er nu muligt at underskrive eller verificere sin besked M, offentlige- og privatnøgle Q, d, samt systemets parameter (H, p, q, g):

DSA-underskrivning	*DSA-verificering*
Input: En besked M og privatnøglen d. Output: En signatur (r, s) for besked M. 1. Vælg $k\ \epsilon\ \mathbb{Z}\{1, \dots, q-1\}$ 2. $r = (g^k\ mod\ p)\ mod\ q$ 3. $s = (k^{-1}(H(M) + d \cdot r))\ mod\ q$ Returner (r, s)	Input: En besked M, en offelig nøgle Q og en signatur (r, s) Output: $Sandt$ eller $Falsk$ 1. verificer at $0 < r < q$ og $0 < s < q$. 2. $w := s^{-1}\ (mod\ q)$ 3. $u_1 := H(M) \cdot w\ (mod\ q)$ 4. $u_2 := r \cdot w\ (mod\ q)$ 5. $v := (g^{u_1} \cdot g^{u_2}\ mod\ p)\ mod\ q$ Returner $Sandt$, hvis $v = r$

4.5. Elliptic-Curve Digital Signature algorithm (ECDSA)

ECDSA-algoritmen følger DSA-algoritmen på mange måder, for både beregningerne af systemparametrene, nøglegenerering og brug af hash-funktioner. Forskellen mellem ECDSA- og DSA-algoritmen, er ECDSA's brug af kurven K og algoritmerne for underskrivning og verificering af signaturer. [23]
Her skal vi dog bemærke at når vi arbejder med elliptiske kurver, bliver operationerne $\oplus$ og $\times$ brugt til henholdsvis, punkt addition og punkt multiplikation.

For ECDSA er parametrene givet ved (H, K, n, G), hvor H er hash-funktionen, den Elliptiske kurve K over finite field $\mathbb{F}_p$ med orden n af generator punktet G.

Til generering af nøgler, gør vi på samme måde som DSA, vi vælger en privatnøgle d, hvor d er inde for intervallet $[1, n-1]$. Derefter kan den offentlige nøgle Q beregnes, med brug af punkt multiplikation.

$$Q = d \times G \tag{8}$$

ECDSA-algoritmen følger DSA algoritmens spor, med få ændringer i underskrivning og verificering af signatur.

ECDSA-underskrivning	*ECDSA-verificering*
Input: En besked M og privatnøglen d. Output: En signatur (r, s) for besked M. 1. Vælg $k \epsilon \mathbb{Z}\,[1, n-1]$. 2. $P := k \times G$ 3. $r = P_x \,(mod\ n)$ 4. $s = k^{-1}(H(M) + dr)\ mod\ n$ Returnerer (r, s)	Input: En besked M, en offelig nøgle Q og en signatur (r, s) Output: $Sandt$ eller $Falsk$ 1. verificer at r og s er inden for $[1, n-1]$ 2. $w := s^{-1}\,(mod\ n)$ 3. $u_1 := H(M) \cdot w\ (mod\ n)$ 4. $u_2 := r \cdot w\ (mod\ n)$ 5. $R := u1 \times G \oplus u_2 \times d$ Returnerer $Sand$, hvis $r = R_x$

Bemærk at P og R repræsentere punkter, hvor P_x og R_x definerer x-koordinatet af punktet

5. Secp256k1 i praksis

Efterhånden er vi blevet introduceret til teorien bag Bitcoin. Både hvordan teknologierne og protokollerne bag Bitcoin hænger sammen i forhold til hinanden. Det er nu muligt at kombinere nogle af disse protokoller og algoritmer til at udøve et kryptografisk system.
Jeg har med brug af det typer baseret programmeringssprog Typescript, udviklet et eksempel på ECDSA-protokollen med Secp256k1's parameter.

Før vi kan generere nøglesæt af offentlige- og privatnøgler og signaturer vi underskrive og verificere, skal vi definere vores parameter for den elliptiske kurve og Finite field. [18]
Secp256k1's parameter får vi $(H, K, \mathbb{F}_p, G, n)$, hvor H er hash-funktionen, K funktion for den elliptiske kurve over Finite field $\mathbb{F}_p$. Hvor parametrene for finite field $\mathbb{F}_p$ er generator punktet G (opdelt i x- og y-koordinater), primtallet p og ordenen n af generator punktet G.
Alle Secp256k1's parameter er defineret således:

$$K(\mathbb{F}_p) = y^2 = x^3 + 7$$

$$Hash = SHA256$$

$$p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$$
$$= \text{FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F}$$

$$n = \text{FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141}$$

$$G_x = \text{79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798}$$

$$G_y = \text{483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8}$$

Når vi skal begynde at implementere diverse ECDSA-algoritmer, kan det kræve meget computerkraft at beregne med 32- og 64 byte nøgler. Dette bliver specielt relevant når vi skal lave udregninger med gange og division af 64-byte offentlige nøgler, med flere gentagelser (f.eks. i punkt multiplikation). De høje bytestørrelser fremkommer af SHA256 hash-funktionens output størrelse på 32-bytes. Disse komplikationer har jeg løst med brug af Montgomery reduktion samt bruge et tredjeparts bibliotek, som er designet til håndtering af større tal. (Se evt. biblioteket her: BN.js)

For at kunne beregne med punkter på den elliptiske kurve K, skal vi først implementere punkt addition og duplikering, samt punkt multiplikation for mere effektive beregninger. Både punkt addition og punkt duplikering tag udgangspunkt i algoritmen fra afsnit 4.2.1, men de er implementeret på forskellige måder for en mere overskuelig kode.

Se bilag for kodeeksempler

(Punkt Addition) Kodeblok 1 - beregner summen af to punkter $\text{point1}, \text{point2}$ *på en elliptisk kurve.*

(Punkt Duplikering) Kodeblok 2 pointDouble beregner dobbelt af et punkt.

(Punkt Multiplikation) Kodeblok 3 pointMul multiplicere et punkt med skalaren k

Ved både funktioner for både punkt addition, multiplikation og duplikerings tjekkes først om de angivet punkter er uendelige $\mathcal{O}$, samt om $point1 = point2$ i additionsfunktionen, da dette afgør om punkt duplikerings-funktionen skal bruges i stedet. Derefter udregner vi λ (repræsenteret som s i koden) som vi bruger til at beregner vores ønsket punkt $(x3, y3)$. Punkt multiplikation bliver k også repræsenteret i bits, for at kunne opnå bedre ydeevne.

Til nøglegenerering finder vi først en privatnøgle d, som vi bruger til at generere den offentlige nøgle Q. Som nævnt i afsnit 4.5 brugte vi punkt multiplikation af privatnøglen d og generatorpunktet G på kurven $(Q = d \times G)$.

(Generering af offentlige nøgle) Kodeblok 4 - beregner en offentlig nøgle med en privatnøgle.

Nu kan vi med vores privat og offentlige nøgler d og Q, underskrive og verificere en besked. Til dette tag vi også udgangspunkt i algoritmerne fra afsnit 4.5.

(Underskrivning af signatur) Kodeblok 5 - vælger en tilfældig integer k *med interval* $[1, n-1]$*, hvis* $k = 0$ *findes en ny* k *værdi. Beregner punkt multiplikation på punkt* r

(Verificering af signatur) Kodeblok 6 - beregner, $R = u1 \times G \oplus u2 \times Q;$
Hvis $R == signatur\ s_r \rightarrow return\ true.$

6. Diskussion & Perspektivering

Hvis man skulle lave et ligene system som Bitcoins, hvordan ville man så gøre det? Ville man bruge RSA eller Elliptic-Curves kryptering? Hvordan skulle blockchainen opsættes? Hvad skal man være opmærksom på, for at sikre et stabilt system uden større risiko for angreb? Det er nogle af de spørgsmål jeg vil prøve at sætte et svar på.

6.1. Elliptic-Curve Discrete Logarith Problem (ECDLP)

Det mest velkendte problem for kryptering og ikke mindst Elliptic-Curve kryptering er ECDLP. Når vi definer en elliptisk kurve K over et Finite field $\mathbb{F}_p$ og defineret P som er et element på $K(\mathbb{F}_p)$ af orden n. *Elliptic-Curve Discrete Logarith Problem* (ECDLP) på kurven K, er givet $Q \in K(\mathbb{F}_p)$ skal heltallet k, $\{0 \leq k \leq n-1\}$ findes så:

$$Q = [k]P$$

Dette betyder at vi skal finde ud af hvor mange gange punkt P skal multipliceres med sig selv, for at finde heltallet k, for at finde Q. ECDLP er en "trapdoor-funktion", da det er svært at finde k når man kun ved Q og P, men nemt at finde Q, hvis P og k vides.

6.2. ECDSA vs. RSA

Både RSA og ECDSA er to meget udbredte asymmetriske algoritmer, men der er stor forskel på hvordan de fungerer og ikke mindst deres nøglegenerering. Med hvilken algoritme ville være bedst at implementere i Bitcoins tilfælde?
Den største forskel mellem RSA og Elliptic-Curve kryptering (ECC), udover hvordan de fungerer, er deres forskel på nøglelængde. ECC er blevet fundet til at have større mængde sikkerhed mod nymoderne metoder, på grund af ECC's kompleksitet. RSA kan forsyne samme niveau som ECC, med derimod med meget længere nøglelængder. Derfor med længere nøgler, vil det tag meget længere tid at knække igennem med et "brute-force attack" (Et angreb, hvor tilfældige kombinationer prøves ind til resultatet, er fundet). En anden fordel som ECDSA tilbyder imod RSA er ydeevne og skalerbarhed. Da ECC giver mulighed for mindre nøgler, tilbyder det samtidig brug af mindre computer- og netværkskraft.

Ofte når der snakkes om sikkerhed i kryptering, bliver sikkerheden beskrevet i bits (også kaldt *bit-security*). Bit-security er et princip som beskriver størrelsesordener af mængden af ressourcer, der er nødvendigt til at bryde en hash-funktion eller andet krypteret data. F.eks. skal en computer bruge 2^{256} beregninger til at bryde en 256-bit security hashfunktion. [25]

Da teknologi bliver bedre og bedre for hvert år, bliver computerkraften også bedre og bedre. Dette betyder også at det bliver nemmere at knække krypteringsalgoritmer med mindre bit størrelser. RSA brugte tilbage i 2007, 1024-bit nøglestørrelser som standart, men blev fordoblet nøglestørrelsen til 2048-bit som standart. Denne fordobling af nøglestørrelser er for at opretholde sikkerheden, medfører større vanskeligheder som kraftigt påvirker RSA's brugbarhed og ydeevne. [26]

ECC	RSA	Beskyttelses levetid
163	*1024*	Til 2010
283	*3072*	Til 2030
409	*7680*	Efter 2031

Tilsvarende nøglestørrelser i bits mellem ECC og RSA [26]

Sikkerhed i bits	RSA Nøglelængde krævet i bits	ECC Nøglelængde krævet i bits
80	1024	160-223
112	2048	224-255
128	3072	256-383
192	7680	384-511
256	15360	512+

Behøvet nøglelængde for at opnå samme sikkerhedsniveau [27]

6.3. Sikkerhed I hashing-funktioner

I afsnit 3.2.1 snakket vi hurtigt om såkaldte ”hash-kollisioner”. Som nævn er en hash-kollision når to forskellige inputs til hash-algoritmen, giver det samme output. Dette kan ske da hash-funktioner har en streng med fast længde, ligegyldigt hvad og hvor lang inputtet er til hash-funktionen. Den faste strenglængde gør det teoretisk muligt for hash-kollisioner at forekomme.
Et angreb mod hash-algoritmer som misbruger sandsynligheden for samme output fra to forskellige inputs, bliver kaldt for et ”birthday attack”.

Et Birthday attack er baseret på *Birthday paradox,* hvor paradokset er den høje sandsynlighed for sandsynligheden at to personer i samme rum af 50 personer deler samme fødselsdag. Ved første glimt ville man regne med at sandsynligheden er 17% da 365 dage på et år, over 50 personer; $\frac{365}{50} = \frac{1}{6} = 17\%$.

Men sandsynligheden på fødselsdagsproblemet er faktisk 97%, da vi også skal sammenligne hver fødselsdag til hver anden fødselsdag. Dette kan vi beregne med følgende formel:

$$\frac{n!}{(n-r)!\,r!} = \frac{50!}{(50-2)!\,2!} = 1.225\ kombinationer \tag{9}$$

Ved brug af denne formel kan vi dermed beregne sandsynligheden for at vu kan finde en hash-kollation i en hash-algoritme. [28]

Bits	**mulige outputs (Hashes)**	**Sandsynlighed for tilfældig hash-kollision i %**									
		10^{-18}	10^{-15}	10^{-12}	10^{-9}	10^{-6}	0.1%	1%	25%	50%	75%
16	2^{16}	< 2	< 2	< 2	< 2	< 2	11	36	190	300	430
32	2^{32}	< 2	< 2	< 2	3	93	2900	9300	50.000	77.000	110.000
64	2^{64}	6	190	6100	190,000	6,100,000	1.9×10^{8}	6.1×10^{8}	3.3×10^{9}	5.1×10^{9}	7.2×10^{9}
128	2^{128}	2.6×10^{10}	8.2×10^{11}	2.6×10^{13}	8.2×10^{14}	2.6×10^{16}	8.3×10^{17}	2.6×10^{18}	1.4×10^{19}	2.2×10^{19}	3.1×10^{19}
256	2^{256}	4.8×10^{29}	1.5×10^{31}	4.8×10^{32}	1.5×10^{34}	4.8×10^{35}	1.5×10^{37}	4.8×10^{37}	2.6×10^{38}	4.0×10^{38}	5.7×10^{38}
384	2^{384}	8.9×10^{48}	2.8×10^{50}	8.9×10^{51}	2.8×10^{53}	8.9×10^{54}	2.8×10^{56}	8.9×10^{56}	4.8×10^{57}	7.4×10^{57}	1.0×10^{58}
512	2^{512}	1.6×10^{68}	5.2×10^{69}	1.6×10^{71}	5.2×10^{72}	1.6×10^{74}	5.2×10^{75}	1.6×10^{76}	8.8×10^{76}	1.4×10^{77}	1.9×10^{77}

Tabellen viser antallet af hash-beregninger der skal til for at opnår en vis sandsynlighed for succesfuld hash-kollation. [28]

6.4. Sikkerhed i blockchain-teknologi

Et distribueret netværk som Bitcoin, skal der være en enighed fra majoritet af netværket før en transaktion og block kan blive tilføjet til den nye blockchain. En mulighed for at misbruge denne funktioner og overtage blockchainen er med et såkaldt ”51% attack”. Et 51% angreb består af en gruppe eller organisation kontrollere mere end 50% af alt valideringskraft i netværket, som gør at gruppen kan ændre i blockchainen da de har flertal. Dette kan forekomme hvis gruppen har nok miners til validere nye blocks, med et flertal og dermed have computerkraft nok til at kontrollere 51% af blockchainen. Dette vil dog kræve rigtig meget økonomisk for at købe computerkraft nok til at udkonkurrere alle de andre computere på det primære netværk. [29]

7. Konklusion

Vi har været igennem rigtig mange algoritmer, protokoller og koncepter som hver spiller en rolle i Bitcoin. Vi har redegjort for hvordan kryptering, Hashing og Blockchainteknologi bruger komplekse matematiske koncepter til at skabe et sikkert miljø i Bitcoin.
Samt har vi redegjort hvordan elliptiske kurver og Finite fields fungere sammen, når både ydeevne og sikkerhed står på dagsorden i Bitcoins blockchain. Samtidig har vi redegjort hvordan Bitcoin håndtere forskellige dilemmaer, hvor vi har analyseret, diskuteret og vurderet Bitcoins kryptering mod den andre krypteringsmetoden RSA. Ved opstillingen af RSA og ECC kunne vi konkludere at nøglelængderne mellem RSA og ECC var den drivene faktor for ECC bedre ydeevne. Til sidst har vi også selv prøvet at implementere ECDSA med brug af Bitcoins secp256k1 parametre i kode, for at se hvordan det hele spiller sammen i praksis.

Bibliografi

[1] S. Nakamoto, »Bitcoin: A Peer-to-Peer Electronic Cash System,« Oktober 2008. [Online].

[2] Ledger, »What is Blockchain Anyway?,« 8 december 2022. [Online]. Available: https://www.ledger.com/academy/what-is-blockchain.

[3] bitcoin.org, »FAQ - Bitcoin,« bitcoin.org, [Online]. Available: https://bitcoin.org/da/faq. [Senest hentet eller vist den 17 december 2022].

[4] I. F. Blake, Elliptic Curves in Cryptography, Cambridge University, 2013.

[5] Website Rating, »websiterating.com,« 9 December 2022. [Online]. Available: https://www.websiterating.com/da/vpn/glossary/what-is-asymmetric-symmetric-encryption/.

[6] Enigma Encoder, »Enigma Encoder,« 29 Maj 2019. [Online]. Available: https://www.101computing.net/enigma-encoder/.

[7] H. Delfs og H. Knebl, Introduction to Cryptography: Principles and Applications, Nürnberg: Springer, 2007.

[8] M. Robshaw , Trapdoor One-Way Function, Springer, 2005.

[9] E. Vestergaard, »RSA-kryptosystemet,« matematikfysik.dk, 2007.

[10] A. Omari, »The Math behind RSA,« 13 september 2022. [Online]. Available: https://towardsdatascience.com/the-math-behind-rsa-910f88b94c36.

[11] Wikipedia, »Indbyrdes primisk,« 5 november 2020. [Online]. Available: https://da.wikipedia.org/wiki/Indbyrdes_primisk.

[12] Wolfram MathWorld, »Greatest Common Divisor,« december 2022. [Online]. Available: https://mathworld.wolfram.com/GreatestCommonDivisor.html.

[13] C. Chung, »2BrightSparks,« December 2022. [Online]. Available: https://www.2brightsparks.com/resources/articles/introduction-to-hashing-and-its-uses.html.

[14] T. v. Werkhoven og W. Penard, On the Secure Hash Algorithm, 2017.

[15] S. Vaudenay, A Classical Introduction to Cryptography, Springer, 2005.

[16] A. Narayanan, J. Bonneau, E. Felten, A. Miller og S. Goldfeder, Bitcoin and Cryptocurrency Technologies, Princeton: Princeton University Press, 2016.

[17] L. Conway, »What Is Bitcoin Halving? Definition, How It Works, Why It Matters,« 4 oktober 2022. [Online]. Available: https://www.investopedia.com/bitcoin-halving-4843769.

[18] bitcoin.it, »Secp256k1,« 24 april 2019. [Online]. Available: https://en.bitcoin.it/wiki/Secp256k1.

[19] W. Stein, Elementary Number Theory, Springer, 2017.

[20] J. Kochert og D. Janett, Points of Finite Order, ETH Zurich, 2020.

[21] ç. k. Koç, »Montgomery reduction with even modulus,« september 1994. [Online]. Available: http://cetinkayakoc.net/docs/j34.pdf. [Senest hentet eller vist den 17 december 2022].

[22] D. Hankerson, A. Menezes og S. Vanstone, Guide to Elliptic Curve Cryptography, Springer, 2004.

[23] N. P. Smart, I. F. Blake og G. Seroussi, Advances in Elliptic curve Cryptography, Cambridge University, 2005.

[24] K. Eisenträger, K. Lauter og P. Montgomery, Fast Elliptic Curve Arithmetic and Improved Weil Pairing Evaluation, University of California & Microsoft Research, 2003.

[25] J. Alwen, »The Bit-Security of Cryptographic Primitives,« AWS Wickr, 13 juni 2017. [Online]. Available: https://wickr.com/the-bit-security-of-cryptographic-primitives-2/. [Senest hentet eller vist den 18 december 2022].

[26] H. Jian, FPGA IMPLEMENTATIONS OF ELLIPTIC CURVE CRYPTOGRAPHY, University of North Texas, 2007.

[27] j. Thakkar, »ECDSA vs RSA: Everything You Need to Know,« sectigostore, 9 juni 2020. [Online]. Available: https://sectigostore.com/blog/ecdsa-vs-rsa-everything-you-need-to-know/. [Senest hentet eller vist den 18 december 2022].

[28] G. Gupta, »What is the Birthday attack?,« The Maharaja Sayajirao University of Baroda, 2015.

[29] Investopedia, »51% Attack: Definition, Who Is At Risk, Example, and Cost,« 28 september 2022. [Online]. Available: https://www.investopedia.com/terms/1/51-attack.asp. [Senest hentet eller vist den 18 december 2022].

[30] D. Forney, Skribent, *Introduction to finite fields.* [Performance]. MIT University, 2005.

Billag

```
public pointAdd(point1: Point, point2: Point): Point {
  // pointAdd computes the sum of two points on the elliptic curve.
  const red = BN.mont(this.p);
  const x1 = point1.x.toRed(red);
  const y1 = point1.y.toRed(red);
  const x2 = point2.x.toRed(red);
  const y2 = point2.y.toRed(red);
  if (this.isInfinity(point1)) {
    return point2;
  }
  if (this.isInfinity(point2)) {
    return point1;
  }
  if (x1.cmp(x2) === 0 && y1.cmp(y2) === 0) {
    return this.pointDouble(point1);
  }
  if (x1.cmp(x2) === 0 && y1.cmp(y2.neg()) === 0) {
    return { x: new BN(0), y: new BN(0) };
  }
  // s = (y2 - y1) / (x2 - x1)
  const s = y2.redSub(y1).redMul(x2.redSub(x1).redInvm());
  // x3 = s**2 - x1 - x2
  const x3 = s.redSqr().redSub(x1).redSub(x2);
  // // computes y3 = s * (x1 - x3) - y1
  const y3 = s.redMul(x1.redSub(x3)).redSub(y1);
  return { x: x3.fromRed(), y: y3.fromRed() };
}
```

Kodeblok 7 PointAdd beregner summen af to punkter point1, point2 *på en elliptisk kurve.*

```
public pointDouble(point: Point): Point {
  const red = BN.mont(this.p);
  const a = this.a.toRed(red);
  const x = point.x.toRed(red);
  const y = point.y.toRed(red);
  if (this.isInfinity(point)) {
    return point;
  }
  const redVal2 = new BN(2).toRed(red);
  const redVal3 = new BN(3).toRed(red);
  // s = (3 * x**2 + a) / (2 * y)
  const s = x
    .redSqr()
    .redMul(redVal3)
    .redIAdd(a)
    .redMul(y.redMul(redVal2).redInvm());
  // x3 = s**2 - 2 * x
  const x3 = s.redSqr().redSub(x.redMul(redVal2));
  // y3 = s * (x - x3) - y
  const y3 = s.redMul(x.redSub(x3)).redSub(y);
  return { x: x3.fromRed(), y: y3.fromRed() };
}
```

Kodeblok 8 pointDouble beregner dobbelt af et punkt.

```
public pointMul(k: BN, point: Point): Point {
  if (k.cmp(new BN(0)) === 0) {
    return { x: new BN(0), y: new BN(0) };
  }
  if (this.isInfinity(point)) {
    return point;
  }
  let q: Point = { x: new BN(0), y: new BN(0) };
  let r: Point = point;
  while (k.cmp(new BN(0)) > 0) {
    if (k.and(new BN(1)).cmp(new BN(1)) === 0) {
      q = this.pointAdd(q, r);
    }
    r = this.pointDouble(r);
    k = k.shrn(1);
  }
  return q;
}
```

Kodeblok 9 pointMul multiplicere et punkt med skalaren k

```
public generatePublicKey(privateKey: BN): BN {
  const pubPoint = this.pointMul(
    privateKey,
    this.concatPoint(this.Gx, this.Gy)
  );
  return this.pointToBN(pubPoint);
}
```

Kodeblok 10 generatePublicKey beregner en offentlig nøgle med en privatnøgle.

```
private sign(
  hashedMsg: BN,
  privateKey: Point,
  preK?: BN | undefined,
): signature {
  // Generate random k (nonce) in interval [1,n-1] where n is order of curve
  let k: BN;
  if (preK) {
    k = preK;
  } else {
    do {
      k = new BN(randomBytes(32), "hex");
    } while (k.eqn(0));
  }
  // r = (x1, y1) = kG
  let point = this.pointMul(k, this.decompressPoint(this.G));
  const r = point.x;
  // s = (x2, y2) = k^-1 * (h + d * r) mod n
  const s = k
    .invm(this.n)
    .mul(hashedMsg.add(r.mul(privateKey.x)))
    .mod(this.n);
  return { r, s };
}
```

Kodeblok 11 sign vælger en tilfældig integer k med interval $[1, n-1]$*, hvis* $k = 0$ *findes en ny k værdi. Beregner punkt multiplikation på punkt r, hvor r's y-koordinat er ignoreret. Beregner* $s = \left(k^{-1} \cdot (h + r \cdot d)\right) mod\ n$.

```
private verify(hashedMsg: BN, signature: signature, publicKey: BN): boolean
  {
  const r = signature.r;
  const s = signature.s;
  if (r.gt(this.n) || s.gt(this.n)) {
    throw new Error("Invalid signature");
  }
  const sInv = s.invm(this.n);
  const u1 = hashedMsg.mul(sInv).mod(this.n);
  const u2 = r.mul(sInv).mod(this.n);
  if (hashedMsg.byteLength() > 32) {
    throw new Error("Hashed message is too long");
  }
  const p1 = this.pointMul(u1, this.decompressPoint(this.G));
  const p2 = this.pointMul(u2, this.decompressPoint(publicKey));
  const R = this.pointAdd(p1, p2).x;
  return r.eq(R);
}
```

Kodeblok 12 verify beregner invers signatur $s \rightarrow sInv = s^{-1}\ (mod\ n)$, $u1 = message \times sInv\ (mod\ n)$, $u2 = r \times sInv\ (mod\ n)$, $R = u1 \times G \bigoplus u2 \times Q$. *Hvis* $R == signatur\ s \rightarrow return\ true$.