Projektrapport:

Automatisk kryptoanalys av bifidchiffer

Program: Civilingenjörsprogrammet i Datorsäkerhet, DVADS19
Kurs: Kryptering 2, Kurskod: MA1491

Författare:
Peter Yliniemi & Timothy Hjort

Innehållsförteckning

Inledning

Målet var att implementera en algoritm som skulle kunna utföra en automatisk, endast känd-klartext attack på ett bifidkrypto. För att konstruera detta behövdes en gedigen kunskap om den matematiska metod som används vid såväl kryptering samt dekryptering av ett bifidchiffer.

Denna rapport presenterar ett program som utför en känd-klartext attack på ett bifidchiffer, vilket framgår av den källkod som bifogas tillsammans med inlämningen. Vidare presenteras en egenskriven definition av hur ett bifidchiffer fungerar, som användes som referens vid implementationen av programmet. Slutligen går rapporten igenom en förklaring av källkoden, resultatet, samt slutsatsen av projektet.

Bifidchiffer - Ett klassiskt kryptosystem

Ett Bifidchiffer är ett kryptosystem som kombinerar transpositions-krypto [1] och användning av en s.k Polybiuskvadrat [2]. Kryptosystemet uppfanns kring år 1901 av Felix Delastelle [2].

Krypteringsmetoden går till på följande vis:

Nyckelkonstruktion

En Polybiuskvadrat skapas, där bokstäver placeras med ett vid tillfället vald ordning (exempelvis slumpmässig ordning), och där **J** och **I** delar samma position i polybiuskvadraten, d v s att **J = I**. Exempel på en ifylld polybiuskvadrat:

	1	**2**	**3**	**4**	**5**
1	**G**	**H**	**O**	**E**	**D**
2	**T**	**Q**	**B**	**R**	**M**
3	**N**	**Z**	**A**	**F**	**W**
4	**S**	**Y**	**C**	**K**	**X**
5	**J**	**P**	**U**	**V**	**L**

OBS: Vi utgår ifrån denna polybiuskvadrat i de övriga exempel i denna sammanställning.

Kryptering steg 1

Låt **M** vara en multimängd sådan att **M = {m1,m2,...,mn-1,mn}** där **m1,m2,...,mn-1** representerar bokstäver i klartexten och **n** är ett positivt heltal.
Då är kardinaliteten av **M** lika med längden av klartexten.
D.v.s om vi låter **n** vara ett positivt heltal sådant att **n** representerar längden av klartexten så har vi att **|M| = n**. Observera här att **M** är en multimängd av den anledning att elementen tillhörande **M** inte behöver vara unika.
Vidare låt mängderna **X = {∀ z ∈ ℤ+ : z <= 5}** och **Y = {∀ z ∈ ℤ+ : z <= 5}** representera varje radnummer respektive varje kolumnnummer i polybiuskvadraten.

Låt **i** ∈ ℤ+ representera platsen för en bokstav i **M.**
För varje **mi** ∈ **M** avläser vi dess rad **xi** ∈ **X** samt kolumn **yi** ∈ **Y** i polybiuskvadraten och skapar multimängderna **K = {x1,x2,...,xn-1,xn}** samt **T = {y1,y2,...,yn-1,yn}**.
Vi observerar att resultatet blir tre multimängder av samma längd, d v s att **|M| = |K| = |T|**.

Exempel:

M = {H,A,P,P,Y,N,E,W,Y,E,A,R}
K = {1,3,5,5,4,3,1,3,4,1,3,2}
T = {2,3,2,2,2,1,4,5,2,4,3,4}

|M| = |K| = |T| = 12

Kryptering steg 2

Vi skapar en ny multimängd **N** bestående av elementpar från **K** och **T** sådana att alla element ur **K** placeras först i **N**, följt efter alla element ur **T**.
Vi har att **N = {(x1,x2),...,(xn-1,xn),(y1,y2),...,(yn-1,yn)}**. Notera att **|N| = |M|**, samt notera att om **|M|** är ett udda tal så blir **N = {(x1,xn-1),...,(xn,y1)(yn-1,yn)}**.

- Exempel:
  K = {1,3,5,5,4,3,1,3,4,1,3,2}
  T = {2,3,2,2,2,1,4,5,2,4,3,4}
  N = {(1,3),(5,5),(4,3),(1,3),(4,1),(3,2),(2,3),(2,2),(2,1),(4,5),(2,4),(3,4)}
  |M| = |K| = |T| = |N| = 12

Kryptering steg 3

För att kryptera klartexten **M** skapar vi en multimängd **C** sådan att varje element i **C** blir en bokstav motsvarande koordinaterna för varje par i **N**. Alltså, vi ser på varje par i **N** som koordinater **(x,y)** i polybiuskvadraten, och skapar **C = {c1,c2,...,cn-1,cn}** där **c1,c2,...,cn-1,cn** är de bokstäver som utläses ur polybiuskvadraten givet koordinaterna ur **N**.
Vi erhåller ett kryptogram **C** av samma längd som **M**.

- Exempel:
  N = {(1,3),(5,5),(4,3),(1,3),(4,1),(3,2),(2,3),(2,2),(2,1),(4,5),(2,4),(3,4)}
  C = {O,L,C,O,S,Z,B,Q,T,X,R,F}
  |M| = |N| = |C| = 12

  D.v.s. krypteras **M** som **C**:
  M = {H,A,P,P,Y,N,E,W,Y,E,A,R}
  C = {O,L,C,O,S,Z,B,Q,T,X,R,F}

Dekryptering

För att dekryptera **C** används samma metod som ovan fast omvänt.
Kort sammanfattat gör vi enligt följande:

1. **N** skapas av koordinaterna för respektive bokstav i **C**
2. **N** delas på mitten, **K** skapas av första halvan och **T** av andra halvan.
3. Vi sätter att **M = ∅.** För varje plats **i** $\in \mathbb{Z}+$ i **K** och **T** skapar vi koordinaten **(xi,yi)** och läser av polybiuskvadraten för bokstaven **mi** som motsvarar denna koordinat. Bokstäverna läggs till i mängden **M** i tur och ordning. Resultatet blir: **M = {m1,m2,...,mn-1,mn}**.

Programmets metodik

Vi noterade att det bästa sätt att göra det var att analysera hur ett bifidchiffer konstrueras och hur det knäcks manuellt, för att sedan kunna få en dator att göra samma sak.

Programmet hämtar först in det hela kryptogrammet som önskas attackeras, samt några mindre kryptogram med tillhörande klartexter. Genom de mindre kryptogrammen och respektive klartexter genereras det vi valt att kalla regler, som har formen **Ax=By där A,B** är bokstäver och **x,y** tillhör **{'r', 'c'}** där **r** och **c** motsvarar rad eller kolumn. Dessa regler visar alltså hur respektive bokstav hör ihop. Om **A** befinner sig på samma rad som **B** vet vi alltså var båda finns om bara en av dem råkar få ett värde.

Reglerna itereras igenom och jämförs med de givna värdena tills inga fler slutsatser kan göras. Samtidigt kontrolleras det även om någon rad/kolumn bara har en plats kvar, och om det finns någon bokstav där vi lyckats bestämma en rad/kolumn som matchar den vi undersöker. Då kan vi helt enkelt sätta nämnda bokstav där då det inte finns några andra möjligheter för den att befinna sig någon annanstans.

En sista slutsats som kan göras är om hela polybiuskvadraten bara har en plats kvar så sätts den sista bokstaven där.
Vid de fallen då inga av de tre försöken till resonemang ger något mer påbörjas en gissningsprocess där programmet tittar på de platser som finns kvar och de bokstäver vars position inte entydigt bestämts och försöker sätta dem på någon av de kvarvarande platserna, valt utefter hur bokstävernas situation ser ut, och en dekryptering samt krypteringsprocess sedan utförs i ett försök att avgöra om de gjorda gissningarna är rimliga. När programmet dekrypterat kryptogrammet samt gjort en testkryptering av samtliga inmatningar, som matchar respektive givna kryptogram, anses den vara klar och kvadraten samt kryptogrammet vi försökt forcera skrivs ut.

Användning av programmet

För att testa programmet packar man upp det i valfri mapp, och kör sedan filen "Bifidcracker.py" i en terminal med kommandot: **"./[SÖKVÄG]/Bifidcracker.py"**.

Programmet kan enkelt attackera andra kryptogram. I sådant fall skriver man in kryptogrammet i **cryptogram.txt**, lägger till de givna bokstäverna i polybiuskvadraten i **values.txt** på formen "**J=5**1" där första siffran representerar raden och andra siffran kolumnen. Övriga bokstäver skrivs på samma sätt med radbyte mellan varje. Sist ändras **input.txt** med dem ledtrådar som finns till kryptogrammet på formen "**yxa#KZU**" där **yxa** är klartexten och **KZU** är kryptogrammet.

Dokumentation av kod

class Bokstav

- En datastruktur som innehåller en bokstav, dess rad och dess column. Enda speciella är __eq__
- __init__
    - Tar in parametrar och sätter bokstav, rad och kolumn. Allt har "?" som default vilket tillåter oss senare att veta om vi har listat ut var bokstaven befinner sig i polybiuskvadraten.
- __eq__
    - Returnerar sant om datastrukturen innehåller samma bokstav som det den jämförs mot.

class Square

- Vi valde att implementera vad vi kallar en "virtuell polybiuskvadrat", där de faktiska raderna och kolumnerna egentligen inte finns i traditionell mening. En simpel implementation hade varit att ha en matris där respektive plats hade varit en bokstav i kvadraten. Vi valde istället att bara ha allt i en hashlista, vilket visserligen försvårar situationen när vi vill skriva ut kvadraten eller dekryptera, då detta innebar snabba uppslag när vi vill hitta en specifik bokstav, då vi bara behöver sätta in bokstaven som nyckel för att få ut dess objekt. Denna implementation gav även fördelen att vi kan utnyttja metrics för att kunna uppskatta antalet bokstäver på rader och kolumner. Detta snabbade upp processer där en rad/kolumn krävde viss status. T.ex. att enbart en plats finns kvar på en rad.
- **__init__**
    - Samtliga metrics sätts till noll. En för varje rad och kolumn. Därefter laddas hela alfabettet, förutom i, in i minnet och objekt av typen Bokstav skapas i en dictionary,

med bokstaven som nyckel. Sedan öppnas filen “values.txt” och de givna värdena sätts i minnet.
    - Tidskomplexiteten för funktionen ökar linjärt med antalet entries i values.txt. Detta då varje entry ger värdet för en bokstav, och denna hittas genast i hashlistan då bokstaven sätts in som nyckel. Så vi har en ordonotation på O(n), där n är antalet entries.

- **getLastpos**
    - Denna funktion utnyttjar att vi använder metrics för att veta raderna och kolumnernas status. Den används för att upptäcka om det finns exakt en plats kvar i polybiuskvadraten genom att räkna antalet metrics som nått värdet ‘**5**’. Om **4** rader och **4** kolumner har ett sådant värde vet vi att det finns exakt en rad och exakt en kolumn som har en ledig plats. Alltså vet vi nu vilken plats som är ledig och att det då bara finns enbart en plats kvar.
    - Om det finns flera platser kvar returnerar den en tuple i formen **(-1, -1)**
- **Setval**
    - Tar in en bokstav och kan ta in en rad eller kolumn. En intern hashlista används för att bedöma om vi redan definitivt satt bokstaven och isåfall görs ingenting men om bokstaven inte är helt bestämd kontrolleras först om rad och/eller kolumn är givet, genom att de annars är **-1**, och isåfall sätts dessa som värde till bokstaven. Därefter kontrolleras om nu bokstaven har både rad och kolumn given och isåfall kastas den in i **TJÄNA** den interna hashlistan så att vi aldrig försöker sätta den igen samt respektive metrics ökas med **1**.
    - Det finns även en felkontroll om relevanta metrics ökat över **5**, isåfall skrivs ett felmeddelande ut.
    - Samtliga operationer har O(1) och därav har funktionen en tidskomplexitet på O(1).
- **Getval**
    - Tar emot en bokstav och returnerar dess rad och kolumn.
-
- **Returnones**
    - Returnerar alla rader och kolumner där respektive metrics är satta till **4**. Dvs de som bara har **1** plats ledig.
- **Fill**
    - Börjar genom att anropa Returnones och därefter går igenom varje rad och tar reda på vilken kolumn i den som var ledig så att man vet den exakta platsen, sedan kontrolleras alla bokstäver tills en hittats där dess rad är satt (och motsvarar den raden vi kontrollerar) samt att kolumnen inte är satt. Eller vice versa. Då sätts den icke-satta koordinaten till rätt värde mha setval så den tomma platsen returneras och sedan påbörjas processen en gång till för varje kolumn.

class bifidcracker

- **Interpret**
    - Tar emot klartext och kryptogram och listar upp vilken bokstav som i klartexten som korrelerar med vilken bokstav i kryptogrammet. Den skapar regler där den indikerar om respektive bokstav har rad eller kolumn gemensamt och sparar dessa i en hashlista som returneras.

- De genererade reglerna har formatet **[Bokstav1]X=[Bokstav2]Y där X,Y ∈ {“r”, “c”}** där **r** representerar rad och **c** representerar kolumn.

- **__init__**
  - De givna klartext+kryptogrammen läses in från values.txt och skickas vidare till interpret. De genererade reglerna sparas sedan internt i en hashlista.
- **Print_square**
  - Skapar en matris och fyller i respektive plats med en bokstav om vi hittat en där rad och kolumn är satt till platsen och går därefter igenom hela matrisen och skriver varje bokstav, eller ett frågetecken om den platsen har okänt innehåll.
- **Checklast**
  - Kontrollerar om det finns enbart en tom plats mha getLastpos och ser därefter om vi har någon bokstav kvar som inte är satt. Isåfall sätts den där mha setval och True returneras. Annars returneras False.
- **checkRelations**
  - Går igenom reglerna som genererats av interpret och ser om någon av dessa samband innebär att en bokstavs rad eller kolumn kan bestämmas. Om så är fallet sätts bokstavens värde mha setval och True returneras. Annars True.
- **Conclusion**
  - Anropar checkRelations, Checklast och Square.fill. Varje gång någon av dessa funktioner returnerar True anropas alla tre igen.
- **Decrypt**
  - Dekrypterar kryptogrammet givet från cryptogram.txt och printar klartexten, med frågetecken på de bokstäver den ej kunnat dekryptera, samt alla platsernas rad och kolumn, med frågetecken på de delar som är okända.
- **Getfree**
  - Returnerar koordinater på samtliga platser där en bokstav inte är satt.
- **Guess**
  - Väljer slumpmässigt en utav de kvarvarande, icke-satta, bokstäverna och försöker sätta en rimlig plats utefter de lediga platser som finns och de parametrarna som är satta.
  - Om varken rad eller kolumn är bestämd väljer den en plats slumpmässigt.
  - Om exakt en av rad eller kolumn är satt så sätts den i en kvarvarande koordinat som matchar den satta parametern.
- **Guessing_game**
  - Kopierar instances av Bifidcracker och börjar därefter gissa de kvarvarande bokstävernas platser mha Guess. Efter en gissning har gjorts kontrolleras det om några problem har dykt upp, ex. Orimliga värden upptäckts av conclusion, och sedan dekrypteras kryptogrammet. Om alla bokstäver i klartexten är bestämda så jämförs kryptogrammet mot ordlistan i ordlista.txt, om en matchning dyker upp så antas kryptogrammet vara dekrypterad. Nyckeln testas sedan mot all inmatning i input.txt där all klartext krypteras mha nyckeln och kontrolleras mot de förväntade kryptogrammen. Om allt går anses nyckeln vara korrekt. Annars körs allting om med nya gissningar.
  - Intressant nog kan den sökta nyckeln variera trots att nyckeln testats mot både kryptogrammet i cryptogram.txt och inmatningen i input.txt.
    - Detta fenomen går att lösa genom att mata in mer klartext/kryptogram i input.txt så nyckeln kan entydigt bestämmas.

Main

- Skapar ett bifidcracker objekt och kör alla relevanta delar. Data läses in, tolkas och nyckeln försöker tas fram. När det inte längre går att definitivt hitta fler tecken i polybiuskvadraten gissas resterande delar. Därefter skrivs kvadrat och det dekrypterade kryptogrammet ut.

Resultat, Reflektion och Slutsats

Programmet har stora skillnader i hastighet beroende på olika körningar och givna tecken. Används uppgiften från ALC kompletteringen i kryptering 1 så går det under en sekund, men tas bara U och L bort från de givna värdena ökar körtiden drastiskt. Detta då antalet nycklar den behöver gissa på skalas utefter antalet tomma platser. Med en lite naiv analys, där vi antar att vi bara får fram de resterande platserna genom gissningar, och antalet tomma platser är n så kan vi kombinatoriskt anse att vi måste välja **1** tecken ur **n** tecken **n** gånger utan återlägg. Ordningen anses viktig då samma bokstav kan sättas på olika platser beroende på vilken gissning vi gör. Matematiskt beskrivs detta bara som **n**. Så vid **2** fria platser har vi **2** möjliga nycklar, **3** fria ger **3** nycklar osv. Om vi nu antar att ett givet värde sätter **3** platser, genom förslagsvis reglerna som genererats, kan vi plötsligt beskriva antalet möjliga nycklar som **3n**. Om vi antar att programmets tidskomplexitet ökar linjärt -- ser vi en snabb ökning i tidsåtgång utefter antalet okända platser.

Ett intressant fall som sker är även att med den inmatning och de testkörningar som gjorts kan nyckeln variera lite vid olika körningar. Detta innebär att samtliga nycklar kan dekryptera kryptogrammet samt klarar alla kontroller. Fenomenet kan troligen enbart motverkas genom att ge mer inmatning och utmatning.

Som slutsats ser vi att programmets funktionalitet kan utökas ytterligare, både för att behöva göra färre gissningar genom att även inspektera kryptogrammet som utsätts för forceringsförsöket så att fler regler kan genereras allteftersom nyckeln bestäms bit för bit, samt genom att implementera stöd till att forcera flera kryptogram då programmet för närvarande är begränsad till en. Dessutom kan tidskomplexiteten i gissningsprocessen behöva förbättras då programmet i nuvarande läge har en snabb tidsökning när givna värden tas bort.

Källhänvisning

[1] Kursmaterial, Introduktion till Kryptologi - Kryptering II, 1.8.3 Transpositionskrypton
Förf: Robert Nyqvist

[2] Automated Ciphertext — Only Cryptanalysis of the Bifid Cipher,
Förf: António Machiavelo och Rogério Reis