Enkla statistiska undersökningar med VIFF

KALLE ARVIDSSON

Examensarbete inom datalogi, grundnivå, 6hp.
Kungliga tekniska högskolan. Skolan för datavetenskap och kommunikation.
Handledare: Mikael Goldmann. Examinator: Mads Dam.

Referat

Secure multiparty computation, MPC, är ett område inom kryptografin, vars mål är att låta flera samarbetande parter ge hemlig indata till en beräkning, som de utför tillsammans. Exempel på tillämpningar är omröstningar där deltagarna inte vill tala om hur de röstat och auktioner där ett bud som inte vinner aldrig blir känt för någon annan än budgivaren.

VIFF är ett ramverk som implementerar olika protokoll för MPC. Denna rapport ger en kort introduktion till MPC, VIFF och VIFF:s protokoll. Dessutom presenteras ett enkelt program som med hjälp av VIFF låter flera deltagare anonymt svara på frågor och sammanställa statistik över alla deltagares svar.

Abstract

Simple statistical surveys with VIFF

Secure multiparty computation, MPC, is a field of cryptography, with the goal to let a set of participants give secret inputs to a computation, which they perform jointly. Examples of applications is votings where the voters don't want to tell anybody how they voted and auctions where a bid which don't win never becomes known to anyone except the bidder.

VIFF is a framework which implements several protocols for MPC. This report gives a short introduction to MPC, VIFF and the protocols of VIFF. It also contributes with a small program, which with the help of VIFF, lets several participants anonymously answer a set of questions and gather statistics an all the answers.

Innehåll

Kapitel 1

Introduktion

Secure multiparty computation[1], förkortat MPC, innebär att flera deltagare tillsammans gör en beräkning, med data som är fördelad mellan deltagarna på ett sätt som gör att den enskilde deltagaren inte känner till den. In- och utdata kan komma från och ges till både dessa deltagare och andra. En deltagare känner bara till den indata han ger, den utdata han får och vad som kan härledas ur dessa. I det ideala fallet är systemet ekvivalent med att låta en utomstående, pålitlig part genomföra beräkningen. Pålitlig och säker innebär här att beräkningen utförs korrekt och att ingenting om indatan avslöjas, förutom vad som kan härledas från utdatan.

Antag att beräkningen gäller en omröstning. Deltagarna vill avlägga röster och få dem räknade, men inte öppet tala om hur de röstat. Ett sätt att utföra omröstningen vore att skriva ner rösterna på många likadana papperslappar och blanda lapparna, kanske med hjälp av en valurna. Valurnan kan i detta fall ses som en utomstående part som garanterar säkerheten. Antag nu att deltagarna inte befinner sig på samma fysiska plats, utan bara kan kommunicera genom ett datornätverk, i värsta och troligaste fallet är nätverket internet. Då fungerar det inte längre att använda en valurna, eller en simulering av en sådan, eftersom deltagarna inte längre kan ha samma kontroll över den, när de inte är närvarande i rummet. En möjlig lösning på problemet är MPC.

En annan tänkbar användning av MPC är en auktion där budgivarna inte vill avslöja sina bud (om budet inte vinner) och inte litar på någon enskild auktionsförrättare. En sådan auktion har faktiskt utförts[3]. 2008 skickade drygt 1000 danska sockerbetsodlare bud för att köpa och sälja odlingskvoter till tre datorer tillhörande forskningsprojektet SIMAP, Danmarks enda sockerproducent

Danisco och danska sockerbetsodlarförbundet DKS. Sedan bestämde programmet som körde på de tre datorerna det optimala priset. Så länge inte två av de tre parterna samarbetade kunde de inte veta vilka bud en odlare lagt. VIFF är en efterträdare till SIMAP och användes för samma auktion 2009.

I [4] föreslås att MPC kan användas för statistiska beräkningar på känslig data.

En grundläggande del av MPC är *Secret sharing*, delning av hemligheter, som låter en hemlighet (hemlig data) "delas", så att flera delar behövs för att återskapa datan. I MPC utförs beräkningar på delarna, medan dessa befinner sig hos olika deltagare.

1.1 Översikt

Det finns flera olika protokoll, som givet en funktion, vanligen i form av en logisk eller aritmetisk formel, utför MPC. Vad ett visst protokoll lämpar sig till begränsas av vilken säkerhet som ges och hur effektivt protokollet är, beräkningen måste gå att genomföra på rimlig tid.

Syftet med arbetet är att ge en introduktion till MPC och beskriva ett par MPC-protokoll och deras egenskaper. Arbetet har utgått från MPC-ramverket VIFF, som implementerar flera protokoll. Två av VIFF:s protokoll beskrivs kortfattat i kapitel 3. En del i arbetet är ett enkelt program som demonstrerar MPC. Programmet använder VIFF och kan därför köras med flera olika protokoll. Det beskrivs i kapitel 4 och 5. I kapitel 6 diskuteras slutligen tillämpningens användbarhet.

Kapitel 2

Bakgrund

2.1 Secret sharing

Secret sharing kan sägas vara kärnan i MPC. Med *secret sharing* kan information lagras av flera deltagare utan att avslöjas för någon enskild av dem. I denna rapport används även den enkla översättningen *hemlighetsdelning*.

En simpel metod för att "dela en hemlighet" är att låta hemligheten, i form av ett tal s i en ring $\mathbb{Z}_m$, vara summan av slumpmässigt valda delar $s_1 \dots s_n$. Delningen utförs genom att välja $s_1 \dots s_{n-1}$ slumpmässigt i $\mathbb{Z}_m$ och sedan välja s_n så att $s = \sum_{i=1}^{n} s_i$. Även om man känner till alla delar utom en, vet man inget mer om hemligheten, eftersom den sista delen, och därmed hemligheten, kan vara vilket tal som helst i $\mathbb{Z}_m$.

En mer intressant metod är *Shamirs secret sharing-schema*, som introducerades av Shamir under den något tvetydiga titeln *How to share a secret*[2]. Med denna metod kan man välja hur många delar som behövs för att återskapa hemligheten, oberoende av antalet delar n. Hemligheten s ett element i en ändlig kropp, till exempel $\mathbb{Z}_p$, där p är ett primtal. Låt $f_s(x)$ vara ett polynom av grad t så att $f_s(0) = s$. Polynomets konstanta term är alltså s. De övriga koefficienterna väljs slumpmässigt. De n delarna är punkter på polynomet, för enkelhets skull väljs $f_s(1), \dots, f_s(n)$.

$$f_s(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_t x^t \tag{2.1}$$

$$a_0 = s \tag{2.2}$$

$$a_1, \dots, a_t \text{ väljs slumpmässigt} \tag{2.3}$$

$$s_i = f_s(i), \text{ för } i = 1, \dots, n \tag{2.4}$$

Med t eller färre delar får man ingen information om hemligheten, men med $t+1$ stycken kan man återskapa den, exempelvis med Lagrange-interpolation [1].

2.2 Secure multiparty computation

Allmänt kan MPC sägas bestå i att n parter utför en beräkning av en funktion $(y_1,\ldots,y_k) = f(x_i,\ldots,x_m)$, med indata från m parter och utdata till k stycken. De tre deltagar-grupperna kan överlappa godtyckligt, exempelvis kan de vara en och samma grupp och utdatan kan vara densamma för alla deltagare. Beräkningen är säker om utdatan är korrekt och information om x_i avslöjas enbart genom utdatan. Dessutom kan man kräva att utdata garanterat ska ges, eller att det råder rättvisa i den meningen att antingen ingen eller alla deltagare får sin utdata.

Säkerhet

Inget MPC-protokoll (för godtyckliga beräkningar) kan garantera en perfekt säkerhet, utan är enbart säkert givet att vissa antaganden håller. På många kryptografiska områden består dessa antaganden vanligen i att en viss beräkning inte kan genomföras på rimlig tid (till exempel att man inte kan faktorisera ett stort heltal, eller beräkna den diskreta logaritmen i en viss grupp). Det finns MPC-protokoll som bygger på sådana antaganden om beräkningar, men de som tas upp i den här rapporten antar istället att man kan lita på en viss andel av deltagarna. Deltagare kan vara opålitliga båda genom att själva vara oärliga, eller genom att någon utomstående fiende övervakar eller kontrollerar dem. Protokollen antar att maximalt ett visst antal deltagare är opålitliga, detta antal betecknas med t. Om det finns fler än t sådana deltagare kan det till exempel vara möjligt för dem att tillsammans avslöja all indata.

Man kan tänka sig olika typer av fiender. Bland annat skiljer man på passiva och aktiva fiender. Den passiva, även kallad halv-ärliga (engelska *semihonest*), fienden kan känna till all data hos de deltagare han attackerat, men kan inte påverka dem så att de inte följer protokollet. Den aktiva fienden har full kontroll över de attackerade deltagarna. I båda fallen kan fienden vara både en yttre fiende eller en oärlig grupp bland deltagarna. Av protokollen som presenteras i kapitel 3, är ett säkert enbart mot en passiv fiende och som har insyn i upp till $t < n/2$ av deltagarna. Det andra är säkert även mot en aktiv fiende, men då med den sämre gränsen $t < n/3$. Givet att antagandet om antalet pålitliga deltagare håller, har protokollen perfekt säkerhet, oavsett hur mycket beräkningar fienden kan göra

påverkas inte säkerheten. Gränserna $t < n/2$ respektive $t < n/3$ är optimala för protokoll med perfekt säkerhet mot passiva respektive aktiva fiender [1].

Man behöver också antaganden om hur kommunikationen fungerar. De två protokollen antar att alla deltagare har parvisa säkra kommunikationskanaler, vilket i praktiken kan uppnås med konventionell kryptering och autentisering (men då är säkerheten inte längre perfekt). Nätverket kan modelleras som synkront eller asynkront[1]. I ett synkront nätverk kommer alla meddelanden som skickas fram inom en viss tidsgräns. Det asynkrona nätverket är en mer realistisk modell, här finns ingen gräns för hur mycket ett meddelande kan försenas. I det senare fallet kan deltagarna inte skilja på ett meddelande som försenats och ett som inte skickats, vilket gör att en fiende enkelt kan orsaka en deadlock.

MPC med Shamirs secret sharing

Båda de protokoll som beskrivs i kapitel 3 använder Shamirs secret sharing. Varje hemligt tal i beräkningen ingår i en hemlighetsdelning. Gradtalet för delningarnas polynom sätts till t, det maximala antalet oärliga deltagare. Då behövs $t+1$ delar för att avslöja ett hemligt tal, vilket hindrar de upp till t opålitliga deltagarna, eftersom de bara har en del var.

Deltagare i ges alltid hemlighetsdelen i, som är $f_s(i)$. Det betyder att han kan addera och multiplicera polynomen i x-värdet i. På så vis blir det trivialt att addera hemliga tal, eller att multiplicera med en konstant. Varje deltagare utför operationerna på sina delar och får en del i en ny delning, vars värde är det sökta resultatet. Följande ekvationer visar hur deltagare i medverkar i en addition av talen x och y och en multiplikation av x med konstanten k.

$$f_{x+y}(i) = f_x(i) + f_y(i) \tag{2.5}$$

$$f_{kx}(i) = k f_x(i) \tag{2.6}$$

Detta fungerar bra för addition, men vid multiplikation av två polynom dubbleras gradtalet, vilket förhindrar att man gör godtyckligt många multiplikationer och sedan återskapar resultatet. De protokoll som beskrivs i nästa kapitel har två olika algoritmer för att utföra multiplikationer. Det ena protokollet börjar med att först skapa en delning med dubbla gradtalet och sedan ur denna skapa en ny delning av samma hemliga tal, fast med rätt gradtal.

Kapitel 3

VIFF

VIFF (*Virtual Ideal Functionality Framework*)[8] är ett programbibliotek, med vilket man kan bygga datorprogram som använder MPC. VIFF är också ett forskningsprojekt och syftar till att vara ett ramverk för implementationer av olika MPC-protokoll. I [5] introducerar Geisler VIFF och dess två första protokoll. Dessa två protokoll beskrivs kortfattat i detta kapitel.

Det ena protokollet är säkert mot en passiv fiende, som korrumperar maximalt $t < n/2$ deltagare. Det andra är säkert även mot en aktiv fiende, men med gränsen $t < n/3$. Båda protokollen bygger på Shamirs secret sharing och kan utföra additioner och multiplikationer. Övriga operationer måste implementeras utifrån dessa. VIFF tillhandahåller algoritmer för jämförelser och logiska operationer (talen `0` och `1` används för sant och falskt).

Den asymptotiska tidskomplexiteten är densamma i båda protokollen, även om det aktivt säkra protokollet är långsammare. Komplexiteten för varje deltagare är $O(n^2|C|k)$, där $|C|$ är beräkningens storlek och k är antalet bitar i det primtals-modulus p som beräkningarna görs med [5]. Om man ser k som en konstant och räknar per multiplikation och tal i utdatan blir komplexiteten $O(n^2)$.

3.1 VIFF:s implementation

VIFF är implementerat i programmeringsspråket Python med nätverks-biblioteket Twisted[9]. Twisted tillhandahåller en klass, `Deferred` (engelska: fördröjd, uppskjuten). Ett `Deferred`-objekt representerar ett värde som ännu inte är känt, till exempel kan det vara data som förväntas komma via nätverket. Man kan koppla ett sådant objekt till en callback-funktion, som anropas då datan blir tillgänglig.

I VIFF ligger datan i `Deferred`-objekt och operationer på den skapar callback-funktioner. Twisted ser sedan till att operationerna utförs när det blir möjligt. Meddelanden till och från de andra deltagarna buffras automatiskt. För att hålla reda på alla tal som skickas mellan deltagarna märks alla meddelanden med en "program-counter", som talar om vilket tal i beräkningen det gäller.

I grunden är nätverket asynkront och protokollen modellerar det också som asynkront, vilket ger en vinst i prestanda och komplexitet. Att simulera ett synkront nätverk innebär att man har synkroniseringspunkter då deltagarna väntar på varandra. Man antar en viss högsta svarstid och om någon blir försenad räknas det som ett fel. Det aktivt säkra protokollet är inte helt asynkront utan har en synkroniseringspunkt.

Kommunikationen krypteras och autentiseras med SSL, på så vis uppfylls i praktiken antagandet om att deltagarna har säkra parvisa kommunikationskanaler, förutsatt att de har kända SSL-certifikat [8].

3.2 Protokoll säkert mot passiv fiende

VIFF:s enklaste protokoll är säkert mot en passiv fiende, som inte korrumperar mer än $t < n/2$ av deltagarna.

Som tidigare konstaterat är det trivialt att utföra additioner, men svårare med multiplikationer, när man använder Shamirs secret sharing. I detta protokoll börjar man med att multiplicera hemlighetsdelarna så att man får en delning med dubbla gradtalet. Låt de två talen och deras produkt betecknas med $xy = z$. Deltagare i beräknar $z_i = x_i y_i = f_x(i) f_y(i)$. $z_1, \ldots, z_n$ utgör en giltig delning av z, men gradtalet är $2t$ istället för t. För att skapa den korrekta delningen av z börjar varje deltagare med att skapa en delning $f_{z_i}(x)$, av sin del z_i. Del i skickas till deltagare i, som får talen $f_{z_1}(i), \ldots, f_{z_n}(i)$. Han betraktar dessa som delar i en hemlighetsdelning och återskapar dess värde. Då får han sin del i en ny delning av z med korrekt gradtal. Att det fungerar beror på att det återskapade värdet är en linjär kombination av delarna, för ett bevis, se [6].

3.3 Protokoll säkert mot aktiv fiende

Detta protokoll använder en annan multiplikationsalgoritm. I förväg skapas delningar av slumpmässiga, hemliga taltripplar a, b, c, som uppfyller $ab = c$, sedan används en trippel för varje multiplikation. Låt $d = x - a$ och $e = y - b$. Produk-

ten av x och y kan då skrivas som i följande ekvation.

$$xy = ((x-a)+a)((y-b)+b) = (d+a)(e+b) = de + ea + db + ab \quad (3.1)$$

d och e är liksom a och b slumpmässiga värden. De offentliggörs för alla deltagarna, varefter delarna för xy kan beräknas enligt ekvation (3.2).

$$f_{xy}(i) = de + ef_a(i) + df_b(i) + c \quad (3.2)$$

VIFF har två alternativa protokoll för att skapa tripplarna. Det ena, *Pseudo-random secret sharing*, har en exponentiell tidskomplexitet, men är det snabbare när antalet deltagare är litet. Gränsen går vid ungefär 10 deltagare enligt Geislers mätningar. Det andra bygger på *hyperinverterbara matriser* och beskrivs i både [7] och [5]. En hyperinverterbar matris är en matris vars alla kvadratiska submatriser är inverterbara. En submatris är en matris där bara element från en delmängd av raderna och kolumnerna ingår.

Nedan ges en intuitiv förklaring till hur delningar av slumpmässiga okända tal skapas med hjälp av en hyperinverterbar matris av storlek $n \times n$. Att metoden är säker och hur taltripplar sedan genereras utifrån de slumpmässig delningarna visas i [7].

För att skapa delningar av $n-2t$ slumpmässiga tal väljer varje deltagare varsitt tal slumpmässigt och fördelar detta bland de övriga med Shamirs secret sharing. Varje deltagare multiplicerar den vektor av delar han mottagit med den hyperinverterbara matrisen. Resultat blir n nya delar. $2t$ av delarna skickas till $2t$ olika deltagare, som kontrollerar att dessa delningar är korrekta, vilket är fallet om olika delgrupper av $t+1$ delar resulterar i samma återskapade värde. Om dessa var korrekta betyder det att också de ursprungliga delningarna var korrekta. De övriga $n-2t$ delningarna är de genererade delningarna. Talen blir slumpmässiga, eftersom åtminstone de $n-t$ pålitliga deltagarna har valt sina ursprungliga tal slumpmässigt.

Genereringen av tripplarna och själva beräkningen sker i två faser. Mellan faserna finns en synkroniseringspunkt, men i övrigt är protokollet asynkront.

Kapitel 4

En tillämpning av MPC

I detta och följande kapitel beskrivs ett enkelt program som med hjälp av MPC utför en statistisk undersökning. Målet är att testa VIFF och att demonstrera MPC, snarare än att göra statistiska undersökningar av hög kvalitet. Då alla undersökningens deltagare också är deltagare i beräkningen, förhindras storskaliga undersökningar, eftersom varje deltagares arbete växer med antalet deltagare, se även stycke 4.4.

Idén är att använda MPC för att göra statistiska undersökningar på deltagarnas villkor. En enskild deltagares svar kommer inte att avslöjas för någon (om protokollet säkerhet håller), vilket är en skillnad jämfört med en typisk undersökning där deltagarens anonymitet bara garanteras av utförarens välvilja. De n stycken deltagarna svarar på m stycken frågor, som de kommit överens om i förväg. Svaren ges som en siffra på en skala. Sedan beräknas medelvärde och annan statistik för svaren på varje fråga och ges som utdata till alla deltagarna.

4.1 Statistiska mått

Här beskrivs de statistiska mått som programmet beräknar. Som källa till dessa refereras till Gunnar Bloms lärobok om statistik[11].

Tre vanliga statistiska mått på *sannolikhetsfördelningar* är förväntat värde $E(X)$, varians $V(X)$, och kovarians $C(X,Y)$ (X och Y är stokastiska variabler). Det förväntade värdet är medelvärdet av alla möjliga värden, viktat efter värdenas sannolikheter. Variansen mäter spridningen för en variabel och kovariansen mäter

beroendet mellan två variabler. Här följer deras definitioner:

$$E(X) = \sum xP(X = x) \text{ för alla } x \tag{4.1}$$

$$V(X) = E((X - E(X))^2) \tag{4.2}$$

$$D(X) = \sqrt{V(X)} \tag{4.3}$$

$$C(X,Y) = E((X - E(X))(Y - E(Y))) \tag{4.4}$$

$$\rho(X,Y) = \frac{C(X,Y)}{D(X)D(Y)} \tag{4.5}$$

Variansen är alltså det förväntade värdet på X avvikelse från $E(X)$ i kvadrat. Ofta använder man standardavvikelsen, $D(X)$, istället för variansen. $D(X)$ har samma enhet som X. Variansen kan ses som ett specialfall av kovariansen, $V(X) = C(X,X)$. Istället för kovarians anges ofta korrelationskoefficienten $\rho(X,Y)$, som alltid är ett tal mellan -1 och 1. Om $\rho(X,Y)$ ligger nära -1 eller 1 indikerar det ett negativt eller positivt samband mellan X och Y. Om det till exempel förhåller sig så att Y alltid avviker uppåt när X avviker neråt, har X och Y negativ korrelation.

För att använda dessa mått på datan kan man tolka deltagarnas svar på en fråga som n stycken stickprov $x_1, \ldots, x_n$, ur en sannolikhetsfördelning X, och uppskatta det förväntade värdet och kovariansen med medelvärdet $\bar{x}$, respektive stickprovs-kovariansen c_{xy}.

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{4.6}$$

$$c_{xy} = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}) \tag{4.7}$$

$$s_x = \sqrt{c_{xx}} \tag{4.8}$$

$$r_{xy} = \frac{c_{xy}}{s_x s_y} \tag{4.9}$$

Att nämnaren i ekvation (4.7) är $n - 1$ istället för n beror på att det då blir en bättre uppskattning av kovariansen (den blir väntevärdesriktig). s_x och r_{xy} är uppskattningar av standardavvikelsen respektive korrelationen.

Att se datan som ett stickprov ur en sannolikhetsfördelning passar bra om man gör en statistisk undersökning och deltagarna är stickprov ur en större population, men även om man bara vill beskriva förhållandet inom gruppen av deltagare kan man använda samma mått.

4.2 Anpassningar för implementationen

Beräkningarna sker i $\mathbb{Z}_p$, men genom att välja ett tillräckligt stort primtal p, så att alla tal i beräkningen är mindre än p, kan man utföra beräkningarna som vanligt. I implementationen sätts p till det minsta primtalet större än 2^{63}, i hopp om att det ska vara tillräckligt stort. Ett alternativ hade varit att förutspå hur stora tal som behövs.

Programmet ska presentera medelvärde och standardavvikelse för svaren på varje fråga och korrelationskoefficienten för svaren på varje par av frågor.

Det är svårt att utföra divisioner i MPC-protokollen. Därför görs bara delar av beräkningarna med MPC. Blanda annat beräknas summan av deltagarnas svar istället för medelvärdet, och delas sedan med n innan den presenteras.

För att beräkna kovarianserna finns flera alternativa formler.

$$c_{xy} = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}) \tag{4.10}$$

$$= \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})y_i \tag{4.11}$$

$$= \frac{1}{n-1}\sum_{i=1}^{n}x_i(y_i - \bar{y}) \tag{4.12}$$

$$= \frac{1}{n-1}\left(\sum_{i=1}^{n}x_i y_i - n\bar{y}\bar{x}\right) \tag{4.13}$$

Eftersom multiplikation mellan två okända tal är den dyraste operationen, är det önskvärt att göra så få sådana som möjligt. Additioner och multiplikationer med konstanter kan ofta bortses ifrån. Skenbart behöver (4.13) $n+1$ multiplikationer, och de övriga n stycken. Det förhåller sig dock så att medelvärdena ska göras kända, så alla formler ger n multiplikationer.

Genom att välja formeln (4.13), behöver enbart följande enkla summor beräknas med MPC:

$$\sum_{i=1}^{n}x_i \text{ för varje fråga} \tag{4.14}$$

$$\sum_{i=1}^{n}x_i y_i \text{ för varje par av frågor} \tag{4.15}$$

Utifrån dessa värden beräknas medelvärden, standardavvikelser och korrelationskoefficienter och presenteras för användaren.

4.3 Kontroll av indata

Deltagarna måste ge sina svar på en skala. Ett sätt att kontrollera att en deltagare inte givit svar som ligger utanför skalan, vore att använda de jämförelseoperationer som VIFF tillhandahåller. Dessa är dock kostsamma, enligt Geislers mätningar[5] är jämförelser 2–3 storleksordningar dyrare än multiplikationer.

Eftersom antalet tillåtna svar är litet, blir följande metod användbar. Antag att svar mellan 0 och 9 är tillåtna. För att kontrollera varje svar x, som någon deltagare ger, beräknas följande:

$$\prod_{a=0}^{9}(x-a) \tag{4.16}$$

Om $0 \leq x \leq 9$ kommer någon av faktorerna och därmed hela produkten att vara noll, annars är den skild från noll. Detta görs till en del av utdatan. Ingen extra information avslöjas om ett giltigt svar, eftersom produkten alltid är noll.

4.4 Komplexitet

Den dyraste operationen är multiplikation mellan två okända tal. För att beräkna kovariansen går det åt n multiplikationer för varje par av frågor, $\binom{m}{2}n = \frac{m(m+1)n}{2}$ stycken, där m är antalet frågor. För att kontrollera indatan går det dessutom åt 9 multiplikationer för varje svar, alltså $9mn$ stycken. Formel (4.17) ger det totala antalet multiplikationer.

$$\left(9m + \frac{m(m+1)}{2}\right)n \tag{4.17}$$

När detta kombineras med att tiden per multiplikation är $O(n^2)$, blir programmets tidskomplexitet $O(m^2n^3)$, eller $O(n^3)$ om man ser antalet frågor som konstant.

Kapitel 5

Implementation

5.1 Förkunskaper i Python

Läsaren som inte förstår programmeringsspråket Python bra nog, hänvisas till dokumentationen [10], men här ges också några förklarande exempel med de inbyggda funktionerna `map` och `reduce`, samt lambda-uttryck och listomfattningar. Till att börja med kan en enkel funktions-deklaration se ut så här:

```
def f(x):
  return x+1
```

`map` är en funktion som applicerar en annan funktion på alla element i en lista. Följande kommer att returnera listan `[2,3,4]`.

```
map(f, [1,2,3])
```

Med ett lambda-uttryck kan en funktion definieras inuti ett uttryck. Följande anrop ger samma resultat som det föregående.

```
map(lambda x: x+1, [1,2,3])
```

`reduce` kan till exempel användas för att beräkna produkten av talen i en lista.

```
reduce(lambda x,y: x*y, [1,2,3])
```

Funktionen som ges som första parameter till `reduce` kommer att appliceras på de två första elementen i listan. Svaret används sedan tillsammans med det tredje

elementet i ännu ett anrop och så vidare till listans slut. Värdet för uttrycket ovan blir alltså `((1*2)*3)`.

En listomfattning kan ses som en `for`-loop där ett värde returneras varje varv och samlas i en lista. Ytterligare ett sätt att beräkna `map(f, [1,2,3])` är:

```
[f(x) for x in [1,2,3]]
```

`mmap` är ingår inte i Pythons standardbibliotek, utan definieras i programmet som kapitlets nästa stycke presenterar. `mmap` fungerar ungefär som `map`, fast applicerar istället en funktion på alla element i en lista av listor (en sådan lista kan ses som en matris).

5.2 Programmet

Här gås de centrala delarna av programmet igenom. Den fullständiga koden återfinns i appendix A.

Alla deltagarna kör instanser av samma program. Variablerna `n` och `m` innehåller antalet deltagare respektive antalet frågor.

Varje protokoll i VIFF är implementerat som en subklass till klassen `Runtime`. Variabeln `runtime` i koden är den aktuella `Runtime`-instansen. `runtime.input` används för att ge indata till beräkningen. `players` är en lista på deltagarna (deras nummer), `Zp` är ett `Field`-objekt, som representerar den ändliga kropp som datan finns i och den tredje parametern är själva datan. Varje anrop till `runtime.input` returnerar en lista med `Share`-objekt, alltså delar av hemligheter. Dessa är även `Deferred`-instanser.

För varje fråga gör varje deltagare ett anrop till `runtime.input` som returnerar en lista med deltagarens delar av alla deltagares svar på frågan. `answers` blir en lista av listor. Vid indexering indikerar första indexet vilken fråga svaret gäller och det andra vilken deltagare.

```
answers = [runtime.input(players, Zp, inputs[i])
             for i in range(m)]
```

Sedan kan beräkningar göras på den delade datan. Addition och multiplikation är omdefinierat för `Share` och resulterar i ett anrop av `addCallback` i `Deferred`. Den första beräkningen kontrollerar att svaren som givits är giltiga, med hjälp av formeln (4.16).

```
#check that an answer is in the allowed range 0..k-1
#the returned value is zero for legal answers
def checkrange(a):
  diffs = map(lambda x: a-x, range(k))
  return reduce(lambda x,y: x*y, diffs)

answerchecks = mmap(checkrange, answers)
```

Sedan beräknas summan av deltagarnas svar för varje fråga, vilket används för att beräkna medelvärdet.

```
sums = map(sum, answers)
```

Här följer det lite mer komplicerade uttrycket som för varje par av frågor och för varje deltagare beräknar produkten av deltagarens svar på frågorna och sedan summerar alla deltagares produkter.

```
# sums of products, to calculate covariances
productsums = [[sum([answers[i][p]*answers[j][p]
                     for p in range(n)])
                for j in range(i+1)]
                  for i in range(m)]
```

Nu när beräkningarna är gjorda (egentligen schemalagda), är det dags att ta fram utdatan och presentera den. `runtime.output` används för att "öppna" värdena. `gather_shares` kombinerar delarna till ett enda `Deferred`-objekt, så att man kan registrera en callback-funktion som körs när alla värden är klara. `presentOutput` beräknar medelvärden, kovarianser, standardavvikelser och korrelationskoefficienter och skriver ut resultatet. `runtime.wait_for` ser till att programmet avslutas när utskrifterna är klara.

```
# replace secret shares with the open values
answerchecks = mmap(runtime.output, answerchecks)
sums = map(runtime.output, sums)
productsums = mmap(runtime.output, productsums)

# gather all shares in the output to a single deferred
# the deferred value is a nested list with all the values
answerchecks = gather_shares(map(gather_shares, answerchecks))
allsums = gather_shares( [ gather_shares(sums),
```

```
    gather_shares(map(gather_shares,productsums)) ] )

answerchecks.addCallback(presentAnswerChecks)
allsums.addCallback(presentOutput)

# wait for the values, and shut down runtime
runtime.wait_for(answerchecks,allsums)
```

Kapitel 6

Diskussion

Denna rapport har gett en överblick av hur MPC och VIFF fungerar och gett ett exempel, inklusive körbar kod, på hur VIFF kan användas. I detta sista kapitel diskuteras denna tillämpning och användningen av MPC i allmänhet.

I alla MPC-tillämpningar bör man fråga sig hur mycket utdatan avslöjar om indatan. Utdatan måste säga något, men inte allt om indatan, annars har man ingen nytta av att använda MPC. Tillämpningen som föreslagits i denna rapport har i en mening en svag koppling mellan in- och utdata. Om ordningen mellan deltagarna och deras indata ändras är irrelevant för utdatan, vilket betyder att det är svårt att koppla en viss deltagare till en viss indata. Samtidigt är kopplingen stark om bortser från deltagarnas ordning, för en given utdata finns det bara ett fåtal möjliga indata. En för tillämpningen intressant fråga är om det finns något alternativ till MPC, eller en enklare beräkning att utföra med MPC, vars resultat är att deltagarnas svar blandas, så att det blir känt vilka svar som har givits, men inte vem som givit vilket.

Det kan konstateras att användningen av MPC för enkäter kräver ovanligt motiverade deltagare. De måste lägga lite extra jobb på att köra programmet, vilket troligen tar lång tid eftersom man måste vänta på att alla deltagare ska svara. Vidare behövs en stark vilja att ge sina svar anonymt, eller eventuellt en vilja att låta andra göra det.

6.1 Tidskomplexitet

I stycke 4.4 konstaterades att programmets körtid växer proportionellt mot antalet deltagare i kubik. Det är förstås en stor begränsning för hur många deltagarna

kan vara.

Programmet har tyvärr inte testats under realistiska förhållanden, men man kan få en antydan till hur lång exekveringstiden kommer att bli genom Geislers mätningar[5]. För till exempel 10 deltagare har Geisler mätt att en multiplikation tar 1.7ms med det "passiva" protokollet och $(10.8 + 2.7)$ms för det "aktiva". De två tiderna för det aktiva protokollet är dels tiden för att skapa en tal trippel, dels tiden för att sedan utföra multiplikationen.

Dessa tider är de, under goda förhållanden, minimala uppmätta tiderna och gäller dessutom enbart multiplikationerna. Så genom att använda dem får man en väldigt optimistisk uppskattning, som mest kan säga till vilken storleksordning tiderna hör. Antag att frågorna liksom deltagarna är 10 stycken. Antalet multiplikationer blir då, enligt formel (4.17),

$$(9 \cdot 10 + 10(10 + 1)/2)\,10 = 1450$$

Multiplicerat med 1.7ms respektive 13.5ms får man tiderna 2.5s respektive 20s. Med 19 deltagare är tiderna per multiplikation är 3.9 och 39.9 ms, vilket med samma beräkningar som ovan ger 11s och 110s, för 10 frågor. Med 31 deltagare blir det 9.5 och 110.7ms för en enda multiplikation, och 43 och 500 sekunder för alla.

Med dessa tider är programmet fullt möjligt att köra, särskilt med tanke på att det är rimligt att anta att man även måste vänta på svaren från de andra deltagarna. Om man väntat på de andra deltagarna i flera timmar eller till och med dagar, kan man troligen acceptera att beräkningen tar en stund. Med ännu fler deltagare blir det dock till slut opraktiskt. En stor statistisk undersökning med tusentals deltagare kommer inte att fungera.

Ett sätt att klara av många deltagare är att bara låta ett fåtal utföra beräkningen. Säkerheten beror då på dessa utvalda deltagare. Hemlighetsdelningarna av indatan kan fortfarande göras av den enskilde deltagaren. Med ett konstant antal deltagare i själva beräkningen, blir körtiden linjärt beroende av antalet deltagare i undersökningen. Detta har även fördelen att de flesta deltagarna slipper att köra huvuddelen av programmet och ha det igång tills alla deltagare givit sina svar. Auktionen som nämndes i introduktionen anordnades på detta sätt, tre parter utförde beräkningen och drygt tusen gav indata. Ett annat sätt att möjliggöra en stor undersökning, vore att göra flera små undersökningar och sedan sammanställa resultaten från dessa.

6.2 Säkerhet

Ett annat problem med att vara tusentals deltagare är att man troligen inte känner alla. Man vet egentligen inte om de andra deltagarna går att lita på. En del av deltagarna kanske inte är riktiga deltagare som bidrar med nyttiga svar, utan kontrolleras av en och samma person, med syftet att avslöja de övriga deltagarnas svar. Förutom att ens säkerhet minskar, blir även utdatan mindre värdefull, då man inte vet om de givna svaren är seriösa. Detta gäller förstås även med färre deltagare, om man inte vet vilka de är.

Säkerhet mot en passiv fiende bör nästan alltid ses som svag, eftersom den i sig inte ger något skydd mot en fiende som är beredd att avvika från protokollet. Även säkerheten som ges av det aktiva protokollet kan ibland ses som låg. Antagandet om att maximalt t deltagare är oärliga, duger troligen inte för data som inte får avslöjas under några omständigheter, men när det gäller svar på en enkät, så kan det dock ofta ses som tillräckligt.

6.3 Slutord

Anonyma undersökningar kan utföras på andra sätt, utan MPC, till exempel kan deltagarna ge sina svar genom någon slags proxy-server. En skillnad är att man med MPC kan ha kontroll över vilka som deltar, samtidigt som man ger dem viss anonymitet. Att ha kunskap om vilka som svarat på en enkät är viktigt. Vid en statistisk undersökning vill man ha kontroll över att urvalet blir slumpmässigt. Vid till exempel en kursenkät, alltså en undersökning som görs bland deltagarna i någon kurs, vill man bara att kursdeltagare ska svara. Just en kursenkät är kanske den troligaste användningen av programmet. I en kurs i kryptografi skulle användningen kunna motiveras inte bara av att man vill göra en bra undersökning, utan också av att man får pröva att använda MPC i praktiken.

Litteraturförteckning

[1] Ronald Cramer, Ivan Damgård, Jesper Buus Nielsen, *Multiparty Computation, an Introduction*. `www.daimi.au.dk/~ivan/smc.pdf`, 2009.

[2] Adi Shamir, *How to share a secret*. Communications of the ACM, volym 22, nummer 11, sidorna 612–613, 1979.

[3] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, Tomas Toft, *Secure Multiparty Computation Goes Live*. Financial Cryptography, volym 5628 av Lecture Notes in Computer Science, sidorna 325–343, Springer, 2009.

[4] Martin Burkhart, Mario Strasser, Dilip Many, Xenofontas Dimitropoulos, *SEPIA: Privacy-Preserving Aggregation of Multi-Domain Network Events and Statistics*. USENIX Security Symposium, Washington, DC, USA, 2010.

[5] Martin Geisler, *Cryptographic Protocols: Theory and Implementation*. Doktorsavhandling, Aarhus Universitet, 2010.

[6] Rosario Gennaro, Michael O. Rabin, Tal Rabin, *Simplified VSS and Fast-track Multiparty Computations with Applications to Threshold Cryptography*. PODC, sidorna 101–111, 1998.

[7] Zuzana Beerliová-Trubíniová, Martin Hirt, *Perfectly-secure MPC with linear communication complexity*. TCC, volym 4948 av Lecture Notes in Computer Science, sidorna 213–230, Springer, 2008.

[8] VIFF Development Team, *VIFF:s Documentation*. `http://viff.dk/doc/index.html`, 2011-02-08.

[9] Glyph Lefkowitz, Twisted Matrix Laboratories, *Twisted*, webbplats inklusive documentation, `http://twistedmatrix.com`, 2011-04-01.

[10] Python Software Foundation, *Python v2.7.1 documentation*. `http://docs.python.org/`, 2011-04-14.

[11] Gunnar Blom, Jan Enger, Gunnar Englund, Jan Grandell, Lars Holst, *Sannolikhetsteori och statistikteori med tillämpningar*. Studentlitteratur, 2005.

Bilaga A

Källkod

Koden för det beskrivna programmet följer på nästa uppslag. Givet att man har lagt koden i en fil med namnet `varians.py`, har skapat konfigurationsfiler för varje deltagare (Se VIFF:s dokumentation [8]), och har en Python 2.x interpreter med namnet `python2` och med tillgång till VIFF och Twisted, kan varje deltagare köra kommandot:

```
python2 varians.py <konfigurationsfil> <svar 1> ...
```

Antalet deltagare definieras av konfigurationsfilerna och antalet frågor av hur många svar deltagarna anger. Programmet antar att alla deltagarna ger samma antal svar.

```
#!/usr/bin/env python2

# Copyright 2008, 2009 VIFF Development Team.
# Copyright 2011 Kalle Arvidsson.
#
# This file is part of VIFF, the Virtual Ideal Functionality Framework.
#
# VIFF is free software: you can redistribute it and/or modify it
# under the terms of the GNU Lesser General Public License (LGPL) as
# published by the Free Software Foundation, either version 3 of the
# License, or (at your option) any later version.
#
# VIFF is distributed in the hope that it will be useful, but WITHOUT
# ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
# or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General
# Public License for more details.
#
# You should have received a copy of the GNU Lesser General Public
# License along with VIFF. If not, see <http://www.gnu.org/licenses/>.

import viff.reactor
viff.reactor.install()
import twisted.internet.reactor
import optparse
import viff.util, viff.config
from viff import runtime, field
from viff.runtime import gather_shares

#Options
oparser = optparse.OptionParser(
            usage="%prog [options] config input ...")
runtime.Runtime.add_options(oparser)
options, args = oparser.parse_args()
Zp = field.GF(viff.util.find_prime(2**63))
passive = False #Set to True to use the passive runtime

#Inputs
id, players = viff.config.load_config(args[0])
inputs = map(int, args[1:])
n = len(players)
m = len(inputs)
k = 10 #Range of the answers is 0..k-1
```

```
print("n = %d, m = %d, my_id = %d" % (n, m, id))
print("my_answers = %s" % inputs)

#map for matrices
def mmap(f,m):
  return map(lambda l: map(f,l), m)

def protocol(runtime):
  print("--- Running protocol ---")

  answers = [runtime.input(players, Zp, inputs[i])
              for i in range(m)]

  #check that an answer is in the allowed range 0..k-1
  #the returned value is zero for legal answers
  def checkrange(a):
    diffs = map(lambda x: a-x, range(k))
    return reduce(lambda x,y: x*y, diffs)

  answerchecks = mmap(checkrange, answers)

  sums = map(sum, answers)

  # sums of products, to calculate covariances
  productsums = [[sum([answers[i][p]*answers[j][p]
                       for p in range(n)])
                  for j in range(i+1)]
                    for i in range(m)]

  # replace secret shares with the open values
  answerchecks = mmap(runtime.output, answerchecks)
  sums = map(runtime.output, sums)
  productsums = mmap(runtime.output, productsums)

  # gather all shares in the output to a single deferred
  # the deferred value is a nested list with all the values
  answerchecks = gather_shares(map(gather_shares, answerchecks))
  allsums = gather_shares( [ gather_shares(sums),
    gather_shares(map(gather_shares,productsums)) ] )

  answerchecks.addCallback(presentAnswerChecks)
```

```
  allsums.addCallback(presentOutput)

  # wait for the values, and shut down runtime
  runtime.wait_for(answerchecks,allsums)

def presentAnswerChecks(inputChecks):
  for q in range(m):
    for p in range(n):
      if inputChecks[q][p] != 0:
        print("Invalid input from player %d on question %d" % (p,q))

#compute correlation coefficient
#using covariance and standard deviations
#return inf if a deviation is zero
def correlation(covariance, sigma1, sigma2):
  if sigma1 == 0.0 or sigma2 == 0.0:
    return float('inf')
  else:
    return covariance/sigma1/sigma2

#prints a list of floats, with fixed column-width
def printList(l):
  print("[" + ", ".join(map("{:5.2f}".format, l)) + " ]")

def presentOutput(allsums):
  [sums, productsums] = allsums

  #extract values from field-elements
  sums = map(int, sums)
  productsums = mmap(int, productsums)

  averages = [float(s)/float(n) for s in sums]

  covariances = [[
    (productsums[i][j] - n*averages[i]*averages[j])/float(n-1)
      for j in range(i+1)]
        for i in range(m)]

  #standard deviations
  deviations = [covariances[i][i]**0.5 for i in range(m)]
  #correlation coefficients
  correlations = [[
```

```
    correlation(covariances[i][j],
                deviations[i], deviations[j])
      for j in range(i+1)]
        for i in range(m)]

  print("averages:")
  printList(averages)
  print("deviations:")
  printList(deviations)
  print("correlations:")
  for row in correlations:
    printList(row)

#Create and start the runtime
if passive:
  from viff.passive import PassiveRuntime
  runtime_class = PassiveRuntime
  threshold = (n-1)//2
  print("Using the passive runtime")
else:
  from viff.active import ActiveRuntime
  runtime_class = ActiveRuntime
  threshold = (n-1)//3
  print("Using the active runtime")
print("threshold = %d" % threshold)
pre_runtime = runtime.create_runtime(id, players,
                threshold, options, runtime_class)
pre_runtime.addCallback(protocol)
twisted.internet.reactor.run()
```