平成22年度

論文題目

nチャンネル通信のための経路制御

神奈川大学 工学部 電子情報フロンティア学科

学籍番号 200703006

小川　真人

指導担当者　木下　宏揚 教授

目 次

図 目 次

表 目 次

第1章

序論

1.1　背景

従来の公開鍵暗号方式では公開鍵の正当性を証明するために認証局のような信頼できる第三者機関が必要であった (図 1.1)。それに対して n チャネルメッセージ伝送方式では事前の鍵が不要なため第三者機関も必要ない。そこで本研究では n チャネルメッセージ伝送方式に着目した。もし n 本のうちの何本かに文書を盗聴・改ざんする敵が潜んでいても、残りの通信路の情報を用いて文書を復号することができる (図 1.2)。今回は n チャンネル通信の要となる、ネットワーク層でのプログラムの実装を目的とした。

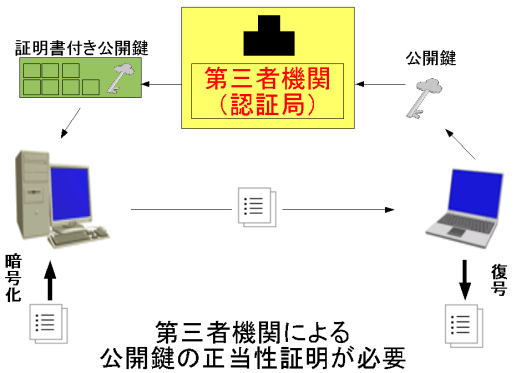


図 1.1　従来の公開鍵暗号方式

図 1.2　n チャネルメッセージ伝送方式

n チャネルメッセージ伝送方式には PSMT(Perfectly Secure Message Transmission) という方式がある。PSMT の安全性の定義は以下のとおりである。

1. 敵は送信メッセージに関する情報を何も得られない。(盗聴耐性)
2. 受信者がメッセージを正しく受信できる確率が 100 % である。(改竄耐性)

また、送信者が受信者に 1 回送信するだけで済む方式を 1-round 方式、送信者と受信者が相互に r 回やり取りを行う方式を r-round 方式と呼ぶ。
PSMT は 1993 年に Dolev ら [1] によって提案された。彼らは、敵が n 本の通信路のうち t 本に潜んでいるとしたときに PSMT プロトコルが存在するための必要十分条件は、1-round 方式では $n \geqq 3t+1$、2-round 方式では $n \geqq 2t+1$ であることを証明した。PSMT においては、$n \geqq 3t+1$ でなければ使えない 1-round 方式よりも、$n \geqq 2t+1$ で使える 2-round 方式のほうが優れている。そこで、1-round 方式における必要十分条件を $n \geqq 2t+1$ に改善するために生まれたのが ASMT(Almost Secure Message Transmission) である。
ASMT の安全性の定義は以下のとおりである。

1. 敵は送信メッセージに関する情報を何も得られない。(盗聴耐性)
2. 受信者がメッセージを正しく受信できる確率が $1-\delta$ 以上である。(改竄耐性)
3. 受信者が正しく受信できない確率が δ であり、そのとき受信者は failure を出力できる。(失敗検知能力)
4. 敵が t 本の通信路を遮断しても受信者は残りの通信路で得た情報だけからメッセージを受信できる。(遮断耐性)

ASMT は 2004 年に Srinathan ら [2] によって提案されたが、そのプロトコルには間違いがあった。その後 2007 年に Kurosawa ら [3] によって厳密に定義された。そのなかで $n = 2t+1$ のときの通信効率の限界が示され、限界に近い通信量で通信できるプロトコルが提案された。上記のプロトコルは計算量が指数関数的であるという問題があった。その問題点を改善し、計算量を多項式時間にした Basic プロトコルを 2008 年に木下研究室が提案した。

1.2 本研究の目的

本論文では、nチャンネルメッセージ伝送に必要な、nチャンネル通信の要となるプログラムの作成と特性の検証を行う。
作成したプログラムを用い、ネットワーク上で動かすことを最終的な目標とする。

第2章

数学的基礎

2.1　合同式

整数環 Z において、整数 m と n を整数 h で割った余りが等しいとき

$$m \equiv n \mod h$$

と記して、m と n とを同一視することを h を法 (modulo h、略して mod h) として合同という。具体例として $h = 5$ である場合、例えば $12 \equiv 7 \equiv 2 \mod 5$ であるから 12、7、2 は 5 を法として同一視される。すなわち合同式を用いると、無数に存在する整数をある整数で割り、同じ余りを有するものどうしを同様の性質をもつ数として分類し、取り扱うことができるのである。

また合同式では整数 a、b、c、d、自然数 m に対して以下のことが成り立つ。

1. $a \equiv a \mod m$ (反射律)
2. $a \equiv b \Rightarrow b \equiv a \mod m$ (対称律)
3. $a \equiv b$、$b \equiv c \Rightarrow a \equiv c \mod m$ (推移律)
4. $a \equiv b$、$c \equiv d \mod m \Rightarrow a + c \equiv b + d \mod m$

   $a - c \equiv b - d \mod m$

   $ac \equiv bd \mod m$
5. もし $ac \equiv bc \mod m$ かつ $gcd(c, m) = 1$ ならば $a \equiv b \mod m$

合同式については [4]、[5] を参照。

2.2 合同式の除法

合同式は加法・減法・乗法・については良い性質を持ち、扱いも簡単である。しかし除法についてはそうではない。そもそも合同式は整数の範囲での記法である。整数を整数で割った場合、整数にならないこともあるので、合同式で除法を考えるときは注意が必要である。例えば、$ab \equiv ac \mod m$ は両辺 a で割って $b \equiv c \mod m$ とすることができるとは限りません。実際 $3 \times 5 \equiv 15 \equiv 9 \equiv 3 \times 3 \mod 6$ ですが、$5 \equiv 3 \mod 6$ ではない。しかし除法を注意深く、上手に解釈すると除算ができる場合もある。そのために、合同式の一次方程式 (一次合同式) を考えてみる。

2.2.1 一次合同式

a、b を整数とするとき、$ax \equiv b \mod m$ となる x を求めることを一次合同式を解くという。合同式の解は m を法として決まるため x が解で $x \equiv x' \mod m$ ならば、x' も解になる。一次合同式は解を持たないこともある。例えば $3x \equiv 1 \mod 6$ は解を持たない。実際この場合 x に解があったとすると、合同式の定義から、$3x - 1$ が 6 で割り切れることになり矛盾となる。解の存在については、次のことが成立する。

- 一次合同式 $ax \equiv b \mod m$ が解 x を持つための必要十分条件は a と m の最大公約数 $gcd(a, m)$ が b を割り切ることである。

[証明]
(必要性)
x を $ax \equiv b \mod m$ の解とする。このとき、$ax - b$ は m で割り切れるので、その商を k とすると、$ax - b = km$ と表される。b と km を移項すると、$ax - km = b$ となり、左辺は $gcd(a, m)$ で割り切れる。したがって、$gcd(a, m)$ は右辺 b を割り切る。

(十分性)
$gcd(a,m)$ は右辺 b を割り切るとし、その商を s とする。このとき $b = s \times gcd(a,m)$ である。ここで後述する拡張ユークリッドの互除法の結果を使うと、

$$ax' + my' = gcd(a,m)$$

となる x' と y' が存在する。この両辺に s を掛けると

$$asx' + msy' = s \times gcd(a,m) = b$$

となる。ここで $x = s \times x'$ と置くと、$ax - b = -msy'$ であるから、$ax \equiv b \mod m$ となる。すなわち、x が解になる。

この証明から解 x は拡張ユークリッドの互除法から計算できることが分かる。上の性質で $b = 1$ の場合が特に重要である。この場合 $gcd(a,m)$ が 1 を割り切るとは $gcd(a,m) = 1$ に他ならない。よって次のことが成立する。

- 一次合同式 $ax \equiv 1 \mod m$ が解 x を持つための必要十分条件は a と m の最大公約数 $gcd(a,m) = 1$ となることである。

この場合、解 x は m を法にして唯一つしかないことも分かる。すなわち、x と x' が共に $ax \equiv 1 \mod m$ の解だとすると、$x' \equiv x \mod m$ となる。実際 $ax \equiv 1 \mod m$、$ax' \equiv 1 \mod m$ とすると、$axx' \equiv x' \mod m$ となる。したがって、

$$x' \equiv axx' \equiv ax'x \equiv 1 \times x \equiv x \ mod\, m$$

となる。このことから m を法とする逆元を定義することができる。

2.2.2 逆元

$gcd(a,m) = 1$ となるとき、一次合同式 $ax \equiv 1 \ mod\, m$ の解 x が m を法にして唯一つ存在する。その x を m を法とする a の逆元という。m を法とする a の逆

元は $gcd(a, m) = 1$ となる a に対してのみ定義される。また m を法とする a の逆元は整数としては唯一つではないが、m を法にした場合は唯一つである。この m を法とする a の逆元 x は整数であるから、他の数に掛けることができる。ここで $ax \equiv 1 \mod m$ であるから、x を掛けることは m を法として考えている限りは a で割ることを意味する。したがって合同式で除法を行う場合は、次のように逆元を用いて行う。

$$a \div b \equiv a \times (b \text{の逆元}) \mod m$$

合同式の除法については [6] を参照。

2.3 ユークリッドの互除法

ユークリッドの互除法は、2つの自然数の最大公約数を導き出すアルゴリズムである。素因数分解に比べて効率よく計算できる。互除法で2つの自然数 a、$b(a > b)$ の最大公約数を見つけるには、次の手続きを用いる。

1. a を b で割り、余り r とする。
2. $r = 0$ の場合は、最大公約数は b であり、手続きは終わりになる。
3. $r \neq 0$ の場合は、a と b の組を b と r に置き換えて、最初の手続きにもどる。

つまり、この1から3の手続きを繰り返して、余りが0になったときに割った数が、最大公約数となるわけである。言い換えれば、余り0を得たときの直前のステップで得た余りが、最大公約数ということになる。
例として、1365と77の最大公約数をユークリッドの互除法で求めてみる。

$$1365 = 17 \times 77 + 56 \quad (\leftarrow 1365 \div 77 = 17 \text{余り} 56 \text{の計算による})$$
$$77 = 1 \times 56 + 21 \quad (\leftarrow 77 \div 56 = 1 \text{余り} 21 \text{の計算による})$$
$$56 = 2 \times 21 + 14 \quad (\leftarrow 56 \div 21 = 2 \text{余り} 14 \text{の計算による})$$

$$21 = 1 \times 14 + \underline{7} \quad (\leftarrow 21 \div 14 = 1 \text{余り} 7 \text{の計算による})$$

$$14 = 2 \times \underline{7} + 0 \quad (\leftarrow 14 \div 7 = 2 \text{余り} 0 \text{の計算による})$$

となり、最大公約数は7となる。
次に互いに素な20と17で、互除法を用いて最大公約数を求めてみる。

$$20 = 1 \times 17 + 3 \tag{2.1}$$

$$17 = 5 \times 3 + 2 \tag{2.2}$$

$$3 = 1 \times 2 + 1 \tag{2.3}$$

$$2 = 2 \times 1 + 0$$

最大公約数は、当然ながら1なので、互除法を使う必要がないように感じられるが、その結果を求める過程の式に大きな利用価値がある。まず式(2.1)、(2.2)、(2.3)を移項して、次の3つの式を得る。

$$20 - 1 \times 17 = 3 \tag{2.4}$$

$$17 - 5 \times 3 = \underline{2} \tag{2.5}$$

$$3 - 1 \times \underline{2} = 1 \tag{2.6}$$

次に式(2.6)の$\underline{2}$に式(2.5)を代入して、3と17に注目してくくる。

$$3 - 1 \times \underline{2} = 3 - 1 \times (17 - 5 \times 3) = 6 \times \underline{3} - 1 \times 17 = 1 \tag{2.7}$$

さらに、式(2.7)の$\underline{3}$に式(2.4)を代入し、20と17に注目してくくる。

$$6 \times 3 - 1 \times 17 = 6 \times (20 - 1 \times 17) - 1 \times 17 = 6 \times 20 - 7 \times 17 = 1$$

この一連の手続きから得た結果を、次のように書き換える。

$$20 \times 6 + 17 \times (-7) = 1$$

上の式は、$ax + by = c$という形になっていて、a、b、c、x、yにあたる数はすべて整数である。このような形の方程式は一次不定方程式といい、整数解のx

とyを求めるものである。つまり、ユークリッド互除法の計算過程を利用することで、$a = 20$、$b = 17$のとき、一次不定方程式の整数解$(x, y) = (6, -7)$が得られるということが示されている。この方法は拡張ユークリッドの互除法と呼ばれ、非常に利用価値の高いアルゴリズムである。一般にaとbを0でない整数とし、aとbの最大公約数をcとすると、一次不定方程式

$$ax + by = c$$

は、整数解(x_1, y_1)を持ち、解の1組は、拡張ユークリッドの互除法を用いて求めることができる。ただし、一次不定方程式の解は、1組だけではない。方程式すべての整数解は、任意の整数kを用いて次のように表される。

$$(x, y) = \left(x_1 + k \cdot \frac{b}{c} \, , \; y_1 - k \cdot \frac{a}{c} \right) \tag{2.8}$$

式(2.8)に示す解の公式を用いれば、一次不定方程式$20x + 17y = 1$のすべての整数解は、次のようになる。

$$(6 + 17k, -7 - 20k) \tag{2.9}$$

$k = -1$の場合、解は$(x, y) = (-11, 13)$である。これを一次不定方程式$20x + 17y = 1$に代入する。

$$20 \times (-11) + 17 \times 13 = 1$$

移項して、式を整える。

$$17 \times 13 = 1 + 11 \times 20 \tag{2.10}$$

式(2.10)をよく見ると、実は、次式と同じ意味であることがわかる。

$$17 \times 13 \equiv 1 \mod 20 \tag{2.11}$$

2.2.2で、$ax \equiv 1 \mod m$の場合、「xはmを法とするaの逆元である」と説明した。すなわち、式(2.11)は20を法にして13が17の乗算に対する逆元であることを意味する。つまり、拡張ユークリッドの互除法を使えば、合同式での逆元が効率よく導き出すことができるのである[7]。

2.4 ラグランジェの補間公式

xy 座標上に k 個の点 $(i, f(i))$ が与えられたとき、それら全てを通る $k-1$ 次関数 $f(x)$ は唯一に定まる。この $f(x)$ を求める方法としてラグランジェの補間公式と呼ばれる次の公式がある。

$$f(x) = \lambda_1(x)f(i_1) + \cdots + \lambda_k(x)f(i_k)$$

ただし

$$\lambda_j(x) = \frac{(x-i_1)\cdots(x-i_{j-1})(x-i_{j+1})\cdots(x-i_k)}{(i_j-i_1)\cdots(i_j-i_{j-1})(i_j-i_{j+1})\cdots(i_j-i_k)}$$

例えば、二次関数 $f(x) = ax^2 + bx + c$ 上の点を $(x_1, f(x_1))$、$(x_2, f(x_2))$、$(x_3, f(x_3))$ として、ラグランジェの補間公式を用いると $f(x)$ は以下のように表せる。

$$f(x) = \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)}f(x_1) + \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)}f(x_2) + \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)}f(x_3)$$

ラグランジェの補間公式については [8] を参照。

第3章

暗号理論の基礎

3.1　公開鍵暗号

公開鍵暗号は、その概念が 1976 年にディフィーとヘルマンによって提案された。具体的な公開鍵暗号化方式の最初のものは、1978 年にリベスト (R.L.Rivest)、シャミア (A.Shamir)、エードルマン (L.Adleman) により提案され、RSA 暗号と呼ばれた。その後、ElGamal 暗号、楕円曲線暗号など多くの公開鍵暗号が発明されたが、現在でも RSA 暗号が最も広く用いられている。

公開鍵暗号を用いる場合には、各ユーザは自分の固有の秘密鍵と公開鍵の対を生成する。いま、A が B に暗号化通信をしたい場合には、B はまず、自分の公開鍵を A に伝える。A はこの公開鍵を用いてメッセージを暗号化し、B に送る。B は自分の秘密鍵を用いて、A から送られた暗号文を復号できる。この方式の利点は秘密鍵がユーザごとに一つだけであるから、その管理が非常に楽になることである。しかし、この公開鍵暗号には大きな問題が二つある。

一つは計算量の問題である。暗号化鍵と復号鍵の非対称性を実現するために、公開鍵暗号の暗号化、復号にはかなりの計算量を要することになる。このため、公開鍵暗号は大量の情報の高速な守秘伝送には向いていない。もう一つは公開鍵の管理の問題である。暗号化の場合でもディジタル署名の場合でも、A

やBの公開鍵が間違いなく本人のものであることが保証されないかぎり、安全な通信や認証が行えない。このような保証を与える仕組みの一つが公開鍵認証基盤 (PKI：public key infrastructure) である。これは認証機関 (CA：certification authority) が各ユーザに発行する公開鍵証明書によって、本人の公開鍵であることの認証を行う方式であり、電子政府の基盤となっている方式である [9]。最後に公開鍵暗号系の中でも重要なものをいくつか簡単に紹介する [10]。

RSA

RSA の安全性は大きな整数の因数分解の難しさに基づいている。

Merkle-Hellman ナップサック

これに関係したシステムは部分集合和問題の難しさに基づいている。しかしながら、様々なナップサック系の全てが危険であることが明らかにされている。(Chor-Rivest 暗号系を除く)

McEliece

McEliece 暗号系は、代数的符号理論に基づいており、いまだ安全であると信じられている。この理論は線形符号の復号問題に基づいている。

ElGamal

ElGamal 暗号系は有限体上の離散対数問題の難しさに基づいている。

Chor-Rivest

これも“ ナップサック ”型暗号系の一種であるが、まだ安全だとみなされている。

3.2 秘密分散共有法

秘密鍵を紛失から守るためには、そのコピーを作って複数の場所に保管しておくことが望ましい。しかし、コピーの数を多くすると盗難の危険が増大してしまう。一方、コピーの数を少なくすると、すべてを紛失してしまう危険が増大する。この相矛盾する二つの問題を解決する方法が、秘密分散共有法である。

秘密分散共有法は、秘密 s の保有者 (ディーラ) と、複数の分散管理者の間で行われる。以降、ディーラを D、n 人の分散管理者を $P_1, \cdots, P_n$ で表す。秘密分散共有法は図 3.1 のように分散段階と再構成段階から構成される。

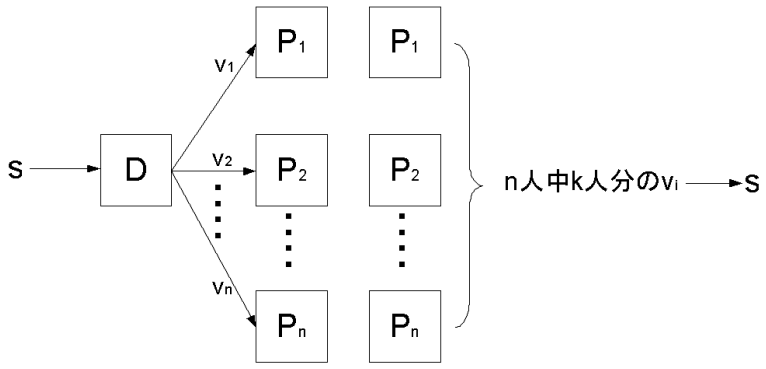


図 3.1 秘密分散共有法

分散段階

図 (a) のように、D は、秘密 s から、n 個のシェアと呼ばれる $v_1, \cdots, v_n$ を計算し、v_i を P_i に与える。

再構成段階

図 (b) のように、n 人の分散管理者のうち何人かが集まり、シェアから s を求める。

(k, n) しきい値秘密分散共有法 (しきい値法) では、$v_1, \cdots, v_n$ は次の条件を満たすように作成されていなければならない。

(条件 1) $v_1, \cdots, v_n$ のうち、任意の k 個から元の秘密 s を復元できる。

(条件 2) どの $k-1$ 個を集めても、s について何も分からない。

秘密分散共有法について詳しくは [8] を参照。

3.3 ハッシュ関数

長いメッセージを短い k ビットに圧縮する関数をハッシュ関数という。ハッシュ関数は、ディジタル署名など、暗号の多くの分野で利用される。

衝突困難なハッシュ関数

ハッシュ関数 H においては、長いメッセージを短くするので、必ず

$$H(x) = H(x')$$

となる衝突ペア (x, x') が存在する。そのような衝突ペア (x, x') を効率よく見つけるのが困難なハッシュ関数を衝突困難なハッシュ関数と呼ぶ。上記のことも含めてハッシュ関数には以下のような安全性が定義されている [10]。

Collision Resistance(衝突困難性)

ハッシュ関数 H は、$x' \neq x$ かつ $H(x') = H(x)$ であるような x と x' を見つけることが計算量的に困難である。

Preimage Resistance(一方向性)

$H(x) = z$ である x を見つけることが計算量的に不可能である。また、このようなハッシュ関数 H を一方向性であるという。

Second Preimage Resistance

x に対して $H(x') = H(x)$ であるような x とは別の x' を見つけることが計算量的に困難である。

固定長圧縮関数

k ビットより長い一定の長さのメッセージを、k ビットに圧縮する関数を固定長圧縮関数と呼ぶ。実際のハッシュ関数の設計においては、長いメッセージに対して固定長圧縮関数を繰り返し適用することにより、可変長の入力に対するハッシュ値を求める、という方針がとられている。

以下に代表的なハッシュ関数について簡単に説明する。

MD4

衝突を有するが、任意の長さの平文を 128 ビットに圧縮する方法で、32 ビットを単位とした演算をする。非常に高速である。

MD5

MD4 をさらに高速にしたものである。

SHA-1

米国政府のシステム調達基準である FIPS 180-1 の中で規定されている代表的なハッシュ関数である。2^{64} ビット未満の任意の長さのメッセージを 160 ビットに圧縮する。

ハッシュ関数については [8]、[10]、[11] を参照。

第4章

nチャネルメッセージ伝送方式

従来の公開鍵暗号方式では公開鍵の正当性を証明するために認証局のような信頼できる第三者機関が必要であった。それに対してnチャネルメッセージ伝送方式では事前の鍵が不要なため第三者機関も必要ない。そこで本論文ではnチャネルメッセージ伝送方式に着目している。

n チャネルメッセージ伝送方式は文書をn 本の通信路を使用してファイルを分散させて通信を行う方式である。分散されたファイルが、すべて違う経路を通り相手に届くことが理想である。しかし、現在のネットワークシステムでは、送信先しか指定できないので、n本の経路を用意するのは不可能である。n本の経路を用意するために、本研究では、経路制御（ソースルーティング）に着目した。nチャネルメッセージ伝送方式にはPSMT と ASMT という方式がある。

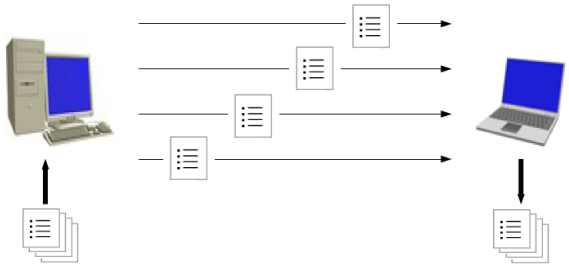


図 4.1　n チャネルメッセージ伝送方式

4.1 PSMT(Perfectly Secure Message Transmission)

4.1.1 PSMTにおける安全性の定義

n チャネルメッセージ伝送方式において次の 2 つの条件を満たしたものを PSMT と呼ぶ。

1. 敵は送信メッセージに関する情報を何も得られない。(盗聴耐性)
2. 受信者がメッセージを正しく受信できる確率が 100 %である。(改竄耐性)

また、送信者が受信者に 1 回送信するだけで済む方式を 1-round 方式、送信者と受信者が相互に r 回やり取りを行う方式を r-round 方式と呼ぶ (図 4.2)。
このとき敵が n 本の通信路のうち t 本に潜んでいるとしたときに PSMT プロトコルが存在するための必要十分条件は、1-round 方式では $n \geqq 3t+1$、2-round 方式では $n \geqq 2t+1$ であることが証明されている。

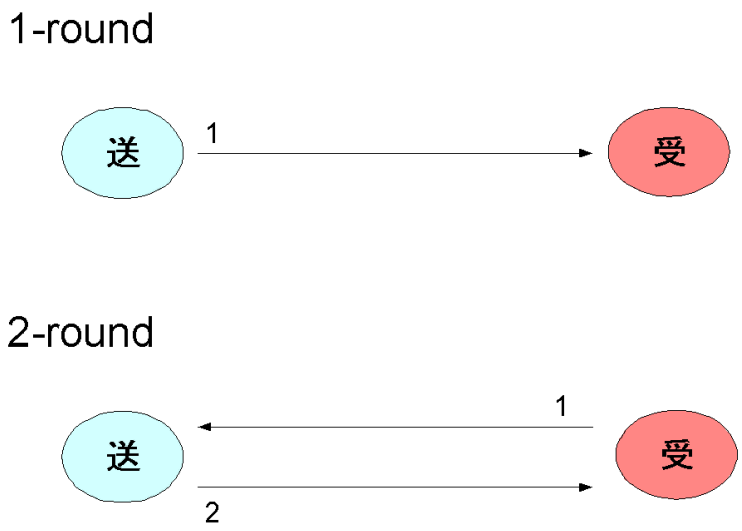


図 4.2 1-round 方式と 2-round 方式

4.1.2 PSMTの歴史

PSMTは1993年にDolevら[1]によって提案された。彼らは、敵がn本の通信路のうちt本に潜んでいるとしたときにPSMTプロトコルが存在するための必要十分条件は、1-round方式では$n \geqq 3t+1$、2-round方式では$n \geqq 2t+1$であることが証明し、また、それぞれの通信量が$O(n)$、$O(2^n)$のプロトコルを提案した。その後2-round方式は1996年にSayeedら[12]が通信量$O(n^3)$のプロトコルを提案し、2006年にはAgarwalら[13]が大量の文書を送ることを前提に通信量$O(n)$で計算量が指数関数的であるプロトコルを提案した。そして2008年にはKurosawaら[14]が通信量$O(n)$、計算量$O(n^3)$となるプロトコルを提案した。このように2-round方式の通信量、計算量は改善されていった。一方、1-round方式は、通信量$O(n)$、計算量が多項式時間となるプロトコルをDolveらが最初に提案しており、既にそれが$n \geqq 3t+1$における最善のプロトコルであった。PSMTにおいては、$n \geqq 3t+1$でなければ使えない1-round方式よりも、$n \geqq 2t+1$で使える2-round方式のほうが優れている。そこで、1-round方式における必要十分条件を$n \geqq 2t+1$に改善するために生まれたのが次に述べるASMTである。

	PSMT	
	2-round	1-round
Dolev 1993	$n \geqq 2t+1$が必要十分条件 通信量：$O(2^n)$	$n \geqq 3t+1$が必要十分条件 通信量：$O(n)$
Sayeed 1996	通信量：$O(n^3)$	
Agarwal 2006	通信量：$O(n)$ ただし大量の文書を送ることが前提 計算量：指数的	
Kurosawa 2008	通信量：$O(n)$ 計算量：$O(n^3)$	

表 4.1 PSMTの歴史

4.2 ASMT(Almost Secure Message Transmission)

4.2.1 ASMT における安全性の定義

ASMT における安全性の定義は以下のとおりである。

1. 敵は送信メッセージに関する情報を何も得られない。(盗聴耐性)
2. 受信者がメッセージを正しく受信できる確率が $1-\delta$ 以上である。(改竄耐性)
3. 受信者が正しく受信できない確率が δ 以下であり、そのとき受信者は failure を出力できる。(失敗検知能力)
4. 敵が t 本の通信路を遮断しても受信者は残りの通信路で得た情報だけからメッセージを受信できる。(遮断耐性)

PSMT と比較して主に異なる点は定義 2 においてメッセージを正しく受信できる確率が $1-\delta$ となっており、失敗した時はそれを検知できるという定義 3 が加わっている点である。

4.2.2 ASMT の歴史

ASMT は 2004 年に Srinathan ら [2] によって提案されたが、そのプロトコルには間違いがあった。その後 2007 年に Kurosawa ら [3] によって厳密に定義された。そのなかで $n = 2t + 1$ での通信効率の限界が以下のように示された。

$$|X_i| \geqq (|S| - 1)/\delta + 1 \tag{4.1}$$

X_i：channel(i) を流れる情報の集合
S：秘密情報の集合
δ：失敗確率

また、Kurosawa らは通信効率の限界に近い通信量で通信できるプロトコルも提案した。そのプロトコルの通信効率は失敗確率を ϵ とすると、

$$|X_i| = \frac{|S|-1}{\delta} + 1 > \frac{|S|-1}{\epsilon} + 1$$

$$\text{ただし } \epsilon = \left\{ \begin{pmatrix} n \\ t+1 \end{pmatrix} - \begin{pmatrix} n-t \\ t+1 \end{pmatrix} \right\} \delta$$

となっている。

4.2.3 Basic プロトコル

Kurosawa らが提案したプロトコルは計算量が指数関数的であるという問題があった。木下研究室がこれを改善して多項式時間となる Basic プロトコルを提案した。

4.2.4 Basic プロトコルの通信量

Basic プロトコルの特徴はハッシュ関数 H を用いる点であった。
敵の Second Preimage Attack が成功したときがこのプロトコルの失敗となる。敵は t 個のハッシュ値に Second Preimage Attack をするので、このプロトコルの失敗確率 ϵ は、

$$\epsilon \leqq [\text{ハッシュ関数 } H \text{ への } \mathit{SecondPreimageAttack} \text{ が成功する確率}] \times t \quad (4.2)$$

ここで H の出力値のビット数を h とすると、出力値は 2^h 通りとなり、H がランダムオラクル (ランダムに値を出力する) と仮定すると H への Second Preimage Attack が成功する確率は $1/2^h$ となる。したがって式 (4.2) は

$$\epsilon \leqq \frac{t}{2^h} \quad (4.3)$$

となる。さらに秘密 s の長さを q ビットとおくと秘密の集合 S との関係は

$$|S| = 2^q \tag{4.4}$$

通信効率の限界式 (式 (4.1)) と式 (4.3)、式 (4.4) より

$$|X_i| \geqq \frac{|S|-1}{\epsilon} + 1 \geqq \frac{2^q - 1}{t/2^h} + 1 = \frac{2^h(2^q-1)}{t} + 1 \tag{4.5}$$

となるので、Basic プロトコルの実際の $|X_i|$ が式 (4.5) の右辺に近ければ通信効率が限界に近いと言える。次にビット数で評価するために両辺の $\log_2$ をとる。

$$\begin{aligned} \log_2 |X_i| &\geqq \log_2 \left\{ \frac{2^h(2^q-1)}{t} + 1 \right\} \approx \log_2 \left\{ \frac{2^h(2^q-1)}{t} \right\} \\ &= \log_2 2^h + \log_2(2^q-1) - \log_2 t \approx h + q - \log_2 t \end{aligned} \tag{4.6}$$

一方、このプロトコルの X_i とはチャンネル ch-i を使って送信者が送る $f(i)$ と $H(f(1))$〜$H(f(n))$ である。秘密 s の長さを q ビットとしているので $f(i)$ も q ビットであり、ハッシュ値は h ビットであるので $|X_i|$ は $q+hn$ ビットとなる。よって式 (5.5) は

$$q + hn = \log_2 |X_i| \geqq h + q - \log_2 t$$

となる。両辺を比較すると左辺の hn と右辺の h との差が大きいことがわかる。したがって Basic プロトコルは通信効率の限界に近いとは言えない。そこで本研究では Basic プロトコルの通信効率を改善したプロトコルを提案する。

第5章

参考研究

この章では、昨年度の栗山 知也氏の Basic プロトコルの通信効率を改善した改良プロトコルを掲載する。

5.1　改良プロトコル

改良プロトコルでは Basic プロトコルと違い、1 度に m 個の s_i を送ることで通信効率を改善している。改良プロトコルの手順は以下のとおりである。
但し、H はハッシュ値であり、安全なハッシュ関数を用いるとする。
$n = 2t + 1$(n：通信路の数、t：敵の数)、送信する秘密情報を s、P を大きな素数とする。

送信者

1. $f_1(x)$〜$f_m(x)$ をランダムで決める。$(f_i(x) = s_i + a_{i1}x + a_{i2}x^2 + \cdots + a_{it}x^t)$

2. $\boldsymbol{F}_1 = f_1(1)\|f_2(1)\|f_3(1)\|\cdots\|f_m(1)$
   $\boldsymbol{F}_2 = f_1(2)\|f_2(2)\|f_3(2)\|\cdots\|f_m(2)$
   $\vdots$
   $\boldsymbol{F}_n = f_1(n)\|f_2(n)\|f_3(n)\|\cdots\|f_m(n)$ とおく。

3. ハッシュ値 $H(\boldsymbol{F}_1)$〜$H(\boldsymbol{F}_n)$ を計算する。

4. 各チャンネル ch-i に $\boldsymbol{F}_i$、$H(\boldsymbol{F}_1)$〜$H(\boldsymbol{F}_n)$ を送る。

敵

- $\boldsymbol{F}_1$〜$\boldsymbol{F}_n$ のうち、t 個しか知らない。
  → $\boldsymbol{F}_i$ 中の $f_i(x)$ は t 次関数なので t 点からは s について何も分からない。
- ここでハッシュ関数 H は一方向性があると仮定するので $H(m)$ から m を逆算できない。
  → ハッシュ値からは s について何も分からない。

受信者

1. $\boldsymbol{F}'_1$〜$\boldsymbol{F}'_n$ を得る。

2. n 本の通信路のうち半分以上の $t+1$ 本は正しい情報であることを利用し、多数決で正しい $H(\boldsymbol{F}_1)$〜$H(\boldsymbol{F}_n)$ を得る。

3. $H(\boldsymbol{F}'_1)$〜$H(\boldsymbol{F}'_n)$ を計算し、$H(\boldsymbol{F}_1)$〜$H(\boldsymbol{F}_n)$ と等しいか調べる。

4. $H(\boldsymbol{F}_i) = H(\boldsymbol{F}'_i)$ となる $\boldsymbol{F}_i$ は $t+1$ 個以上ある。
   → ラグランジェの補間公式を用いて $f_i(x)$ を復元し、s を得られる。

5. もし $f_i(x)$ が復元できなければ failure を出力する。

5.1.1 改良プロトコルの計算量

改良プロトコルでは 1 度に m 個の秘密を送るので、Basic プロトコルの約 m 倍の計算量が必要である。しかし、$m = O(n)$ であれば多項式時間のものを m 倍しても多項式時間であるので、改良プロトコルの計算量は多項式時間である。

5.1.2 改良プロトコルの通信量

改良プロトコルでは1度にm個の秘密sを送るのでsの長さはqmビットとなる。また、使用するハッシュ関数は変わらないのでハッシュ関数Hの出力値はhビットのままである。

Basicプロトコルの通信効率の限界式(4.5)は

$$|X_i| \geqq \frac{|S|-1}{\epsilon} + 1 \geqq \frac{2^q - 1}{t/2^h} + 1 = \frac{2^h(2^q-1)}{t} + 1$$

であった。

通信効率の限界式より

$$|X_i| \geqq \frac{|S|-1}{\epsilon} + 1 \geqq \frac{2^{qm} - 1}{t/2^h} + 1 = \frac{2^h(2^{qm}-1)}{t} + 1$$

となる。ビット数で評価するため両辺の$\log_2$をとると

$$\begin{aligned} \log_2 |X_i| &\geqq \log_2 \left\{ \frac{2^h(2^{qm}-1)}{t} + 1 \right\} \approx \log_2 \left\{ \frac{2^h(2^{qm}-1)}{t} \right\} \\ &= \log_2 2^h + \log_2(2^{qm}-1) - \log_2 t \approx h + qm - \log_2 t \end{aligned}$$

一方、このプロトコルの$|X_i|$とはチャンネルch-iを使って送信者が送る$\boldsymbol{F}_i$と$H(\boldsymbol{F}_1)$〜$H(\boldsymbol{F}_n)$である。ここで$\boldsymbol{F}_i$はqmビットであり、$H(\boldsymbol{F}_1)$〜$H(\boldsymbol{F}_n)$はhビットなので

$$qm + hn = \log_2 |X_i| \geqq h + qm - \log_2 t$$

$q = h$、$m = n$とすると

$$2hn = \log_2 |X_i| \geqq h + hn - \log_2 t > hn - \log_2 n$$

オーダで評価すると

$$O(hn) = O(\log_2 |X_i) \geqq \Omega(hn - \log_2 n)$$

ここでnを十分に大きい値だとすると

$$O(hn) = O(\log_2 |X_i) \geqq \Omega(hn - \log_2 n) \to \Omega(hn)$$

となり、オーダで考えると等しいことがわかる。したがって改良プロトコルは通信効率の限界に近いと言える。

5.2 プログラム

改良プロトコルをプログラム実装した結果、正しく動作した。
最終的な目標をネットワーク上での実装としているため、それに伴う様々な準備を行った。

5.2.1 作成したプログラムの構造

今回プログラムの作成はFlash(ActionScript)を使用した。Flashは暗号関係のライブラリ(as3crypto[15]参照)が充実しており、これにより多倍長とハッシュ関数の導入が可能になった。流れとしてはmainファイルで全体を管理し、処理を各オブジェクトで行う。送信側の処理をSendObject、受信側をReceiveObject、改竄をFalsifyObjectの三つに分割している。

暗号化から復号までの流れ

mainファイルで任意bit数の素数Pを生成する。
↓
送りたい情報Sの長さをm個に分割
↓
送信オブジェクトで送信セットを作成
↓
改竄オブジェクトで送信セットを改竄
↓

受信オブジェクトで受信した情報を復号

↓

復号された情報を表示

その構造を下図に示す。

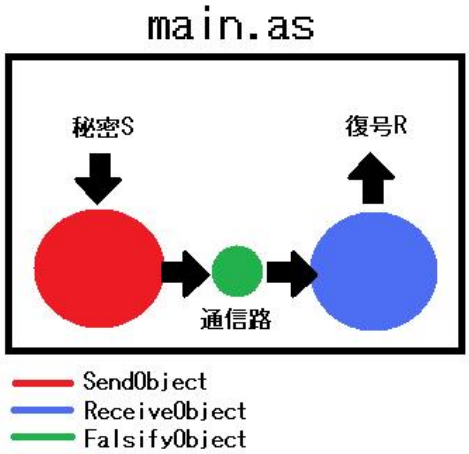


図 5.1 内部構造のイメージ

5.2.2 各オブジェクトの構造

継承した新しいクラスは作らずにBigIntegerクラスを使用した。

送信側オブジェクトの手順

1. 1 〜 9行目 ランダム係数aを生成
2. 11〜46行目 素数Pとaを使い$f(x)$を作成
3. 48〜59行目 ハッシュ値を計算
4. 61〜78行目 F_iとハッシュ値で送信セットを作成

ソースコード 5.1 SendObject

```
1       //ランダム係数a[i][j]の生成
2
3       for( i=0;i<S_loop;i++ ) {
4               a[i] = new Array(2); //2次元配列に
5               a[i][0] = new BigInteger(GenerateKey(256));//256 bitのランダム数生成
6               a[i][0] = a[i][0].mod(P); //配列はメソッドが検出されない
7               a[i][1] = new BigInteger(GenerateKey(256));//256 bitのランダム数生成
8               a[i][1] = a[i][1].mod(P);
9       }
10
11      //Fs_send[i][j]の作成
12
13      for( i=0;i<S_loop;i++ ) {
14              S_bigpiece[i] = new BigInteger(S_piece[i]);
15              //BigIntegerに変換して代入
16      }
17
18      for( i=0;i<S_loop;i++ ) {
19              Fs_send[i] = new Array(5); //Fs_sendを2次元配列に
20      }
21
22      for( i=0;i<S_loop;i++ ) {
23              for( j=0;j<5;j++ ) {
24                      g = new BigInteger(j.toString()); //gにjを代入
25                      BItemp0 = new BigInteger("1");
26                      g = g.add(BItemp0); //gを+1しておく(1~5で使いたい為)
27
28                      BItemp0 = a[i][0]; //a[0]
29                      BItemp1 = BItemp0.multiply(g); //a[0]*g
30
31                      BItemp0 = a[i][1]; //a[1]
32                      BItemp2 = g.multiply(g); //g*g
33                      BItemp2 = BItemp2.multiply(BItemp0); //a[1]*g^2
34
35                      BItemp0 = new BigInteger("0"); //f(g) = 0
36                      BItemp0 = BItemp0.add(BItemp1); //f(g) = a[0]*g
37                      BItemp0 = BItemp0.add(BItemp2); // f(g) = a[0]*g + a[1]*g^2
38                      BItemp0 = BItemp0.add(S_bigpiece[i]);
39                      // f(g) = S + a[0]*g + a[1]*g^2
40
41                      BItemp0 = BItemp0.remainder(P);
42                      // f(g) = S + a[0]*g + a[1]*g^2 mod P
43
44                      Fs_send[i][j] = BItemp0;
45              }
46      }
```

```
47
48      //ハッシュ値H_send[i]の生成
49
50      for( i=0;i<5;i++ ) {
51              H_string[i] = Fs_send[0][i].toString();
52              if ( S_loop > 1 ) { //2つ以上に分割されているかどうか
53                      H_string[i] = H_string[i].concat(Fs_send[1][i].toString());
54                      if ( S_loop > 2 ) { //3つ以上に分割されているかどうか
55                              H_string[i] = H_string[i].concat(Fs_send[2][i].toString());
56                      }
57              }
58              H_send[i] = GenerateSHA1(H_string[i]);
59      }
60
61      //送信セットの作成
62
63      total = 5 + S_loop; // ハッシュの個数 + 分割個数
64
65      for( i=0;i<total;i++ ) {
66              F_send[i] = new Array(10); //F_sendを2次元配列に
67      }
68
69      for( k=0;k<5;k++ ) { //5セット作る
70              for( i=0;i<S_loop;i++ ){
71                      F_send[k][i] = Fs_send[i][k];
72                      //送信セットの頭からFs_sendを入れていく
73              }
74              for( j=S_loop;j<total;j++ ) {
75                      F_send[k][j] = H_send[j-S_loop];
76                      //Fs_sendの次にハッシュ値を入れる
77              }
78      }
```

受信オブジェクトの手順

1. 12〜 34 行目 多数決で正しい $H(F_1)$〜$H(F_n)$ を決める
2. 49〜 74 行目 ハッシュ値を自身が計算したハッシュ値と比較する
3. 76〜106 行目 ラグランジェの補間公式で復号する
4. 108〜114 行目 復号された m 個の情報を結合して目的の情報 S を得る

ソースコード 5.2 ReceiveObject

```
1       for ( i=0;i<total;i++ ) {
2               F_receive[i] = new Array(10); //F_receiveを2次元配列に
3       }
4
5       for( i=0;i<5;i++ ) {
6               for( j=0;j<total;j++ ) {
7                       F_receive[i][j] = F_send[i][j];
8                       //受信セットに改竄した送信セットを代入
9               }
10      }
11
12      //多数決で正しいH_receive[i]を決定
13
14      j = 1;
15      i = 0;
16      k = 0;
17      flag = 0;
18      while( j < 5 ) {
19              flag = 0;
20              for( j=S_loop;j<total;j++ ) {
21                      for( k=i+1;k<5;k++ ) {
22                              if( F_receive[i][j] == F_receive[k][j] ) {
23                                      flag++;
24                              }
25                      }
26                      if ( flag >= 2 ) {
27                              H_receive[j-S_loop] = F_receive[i][j];
28                      }else{
29                              i++;
30                              j--;
31                              flag = 0;
32                      }
33              }
34      }
35
36      //f_receive[i][j]を取り出す
37
38      for( i=0;i<total;i++) {
39              f_receive[i] = new Array(10); //f_revceiveを2次元配列に
40      }
41
42      for( i=0;i<S_loop;i++ ) {
43              for( j=0;j<5;j++ ) {
44                      f_receive[i][j] = F_receive[j][i];
45                      //受信セットからf(g)を取り出す
46              }
47      }
```

```
48
49      //取り出したf_receiveのハッシュ値を計算
50
51      for( i=0;i<5;i++ ) {
52              H_string[i] = f_receive[0][i].toString();
53              //H_stringに分割した一つ目を代入
54              if ( S_loop > 1 ) { //2つ以上に分割されているかどうか
55                      H_string[i] = H_string[i].concat(f_receive[1][i].toString());
56                      //H_stringに分割した二つ目を追加
57                      if ( S_loop > 2 ) { //3つ以上に分割されているかどうか
58                              H_string[i] = H_string[i].concat(f_receive[2][i].toString());
59                              //H_stringに分割した三つ目を追加
60                      }
61              }
62              H_calculate[i] = GenerateSHA1(H_string[i]);
63              //f[g]を繋げたH_stringをGenerateSHA1()に代入してハッシュ値を得る
64      }
65
66      //ハッシュ値の比較
67
68      flag = 0;
69      for( i=0;i<5;i++ ) {
70              if( H_receive[i] == H_calculate[i] ) {
71                      select[flag] = i + 1;
72                      flag++;
73              }
74      }
75
76      //復号処理 ラグランジェの補間公式
77
78      for( i=0;i<S_loop;i++) {
79              select[0] = new BigInteger(select[0].toString());
80              select[1] = new BigInteger(select[1].toString());
81              select[2] = new BigInteger(select[2].toString());
82              BItemp5 = new BigInteger("0");
83              //初期化しておかないと足し算で足してしまう
84              for( k=0;k<=2;k++ ) {
85                      BItemp3 = new BigInteger("1");
86                      //1番目のループの掛け算で0にならないように代入
87                      for( j=0;j<=2;j++ ) {
88                              if( j!=k ) {
89                                      BItemp0 = modsubtract(BIzero,select[j],P);
90                                      // x - x2
91                                      BItemp1 = modsubtract(select[k],select[j],P);
92                                      // x1 - x2
93                                      BItemp2 = moddivide(BItemp0,BItemp1,P);
94                                      // ( x - x2 ) / ( x1 - x2 )
```

```
95                                      BItemp3 = modmultiply(BItemp3,BItemp2,P);
96                                  // (( x - x2 ) / ( x1 - x2 )) * (( x - x3 ) / ( x1 - x3 ))
97                                  }
98                          }
99                          BItemp4 = modmultiply(f_receive[i][select[k]-1],BItemp3,P);
100 // BItemp4[0] = f(x1) * (( x - x2 ) / ( x1 - x2 )) * (( x - x3 ) / ( x1 - x3 ))
101                         BItemp5 = modadd(BItemp5,BItemp4,P);
102                  // BItemp4[0]+BItemp4[1]+BItemp4[2]
103             }
104             decryption[i] = BItemp5;
105             //次のループでBItemp5が初期化されるので保管
106     }
107
108     S_decryption = decryption[0].toString();
109     if ( S_loop > 1 ) {
110             S_decryption = S_decryption.concat(decryption[1].toString());
111             if ( S_loop > 2 ) {
112                     S_decryption = S_decryption.concat(decryption[2].toString());
113             }
114     }
```

改竄オブジェクトの手順

1. 1〜17行目 敵の潜んでいる経路をランダムに決定
2. 20〜45行目 同じ長さのランダムな文字列に置き換える

ソースコード 5.3 FalsifyObject

```
1       //敵の位置決定
2
3       i = 0;
4       flag = 0;
5       while ( i<2 ) {
6               flag = 0;
7               attack[i] = Math.floor(Math.random()*100)%5; //1~5までのランダムな数
8               for( j=0; j<i; j++ ) {
9                       if ( attack[j] == attack[i] ) {
10                              flag = 1;
11                      }
12              }
13              if ( flag == 0 ) {
14                      i++;
15              }
16
17      }
```

```
18
19
20      //改竄
21
22      for ( i=0;i<total;i++ ) {
23              F_receive[i] = new Array(10); //F_receiveを2次元配列に
24      }
25
26      for( i=0;i<2;i++ ) {
27              for( j=0;j<total;j++ ) {
28                      F_send[attack[i]][j] = new BigInteger(GenerateKey(160)).mod(P);
29                      //敵の位置に変数total分160bitを生成して代入
30              }
31              for ( k=0;k<S_loop;k++ ) {
32                      F_send[attack[i]][k] = new BigInteger(GenerateKey(256)).mod(P);
33                      //敵の位置に変数S_loop分62文字を生成して代入
34              }
35      }
36
37      for ( i=0;i<total;i++ ) {
38              F_receive[i] = new Array(10); //F_receiveを2次元配列に
39      }
40
41      for( i=0;i<5;i++ ) {
42              for( j=0;j<total;j++ ) {
43                      F_receive[i][j] = F_send[i][j]; //受信セットに改竄した送信セットを代入
44              }
45      }
```

5.3 ネットワーク上での実装

本研究はネットワーク上での実装を最終目標としている。しかし、いきなり既存のネットワーク上で実装することはプロトコルの仕様以外の部分でも問題が生じる。そのため段階的に実装していく必要がある。次の段階として仮想ネットワーク上での実装を目指す。

5.3.1 仮想ネットワークの利点

次の段階として仮想ネットワークでの実装を目指す理由（利点）は実際のインターネット構造を無視できることにある。仮想ネットワークの場合既存のネットワークと比べてトラフィックや経路障害もない。それに加えて経路決定の問題（後述）をある程度緩和することができる。

5.3.2 実装上の問題

実装する上での問題は

1. 通信プロトコル
2. 経路決定の問題

である。通信プロトコルはTCP/IPの上に乗せるソフトウェアは何を使うのか、といった問題。経路決定の問題というのはプロトコルの仕様上n本それぞれ別々の経路を通る必要があるが、ルータは自身の持つ経路表に基づいた最適な経路を選択するため経路が全て一緒になってしまうことである。

5.3.3 問題の解決案

データを送る方法はFlashを使用する。Flashは容易にXMLにアクセスするメソッド等が用意されているためこれを利用する。またFlashはWEBベースであるためOSへの依存度が低く、既存の仮想ネットワーク構築ソフト上での動作が期待できる。サーバ（受信者）側にXML形式のファイルを経路の数だけ用意し、そこにクライアント（送信者）側がデータ（文字列）を書き込む。これによりサーバ（受信者）は必要な分のXMLファイルを使って復号することができる。経路決定の問題はソースルーティングの技術を使うことにより解決する。ソースルーティングとはルータが普段が自動的に選択している経路

を送信者が明示的に指定することができる技術である。途中のすべての経路を指定する「ストリクトソースルーティング」といくつかの経路を通過することを指定して、それ以外は途中のルータに任せる「ルーズソースルーティング」の2種類があるが、ここではストリクトソースルーティングを使う。実際のインターネットは大規模なネットワークであるため設定が困難であるが、仮想ネットワークでは最小構成でTCP/IPのネットワークを組むことができるため容易に使用できる。これにより経路が目的地に近付くにつれて一緒の経路を通ってしまう可能性を減らすことができる。一緒の経路を通ることはセキュリティ面において非常に危険であり、第三者に復号される可能性が高くなる。最小構成を図5.2、実際のインターネットを図5.3に示した。

5.3.4 実装計画

まず初めに仮想ネットワークを構築するソフトを決定する。選択する上での条件としては

1. 複数のホスト、ルータが設置できる
2. 設置したホストそれぞれでWebが使用可能
3. ソースルーティングに対応している

ことが必要である。次に、サーバ-クライアントでXMLファイルを読み書きするプログラムを作成する。そして、今回作った改良プロトコルにXMLファイルから文字列を読み出し復号するオブジェクトに代入する改良を加える。最後にソースルーティングによって経路指定を実装する。Linuxであれば「ポリシールーティング」などの機構を利用する。「ポリシールーティング」とは指定条件のパケットに特定のルーティングを行う技術であり、ソースルーティングとも呼ばれている。

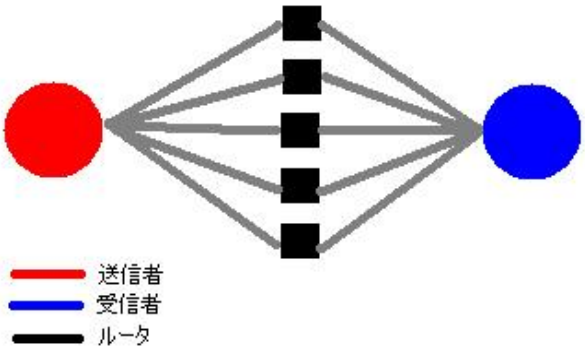


図 5.2 仮想ネットワークのイメージ

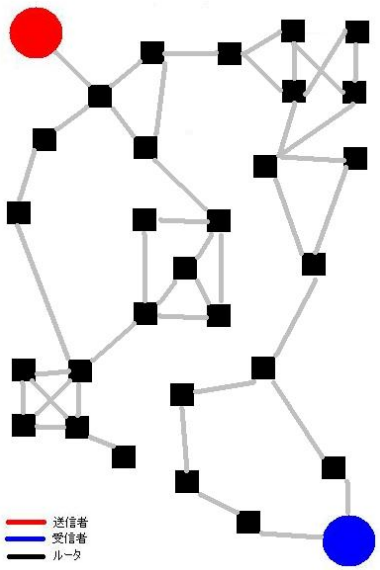


図 5.3 実際のネットワークのイメージ

第6章

経路制御（ソースルーティング）

この章では、経路制御、ヘッダーと実験環境について説明する

6.1　ルーティング

インターネットのように大規模なネットワークでは、あて先ネットワークまでデータを送信したい場合、複数の経路があることが大半である。そこで、パケットの中継地点となるルータが、適切だと思われる経路にパケットを送り出し、あて先ネットワークまでデータを届けるのである。このように、通信経路から最適な経路を選択・制御する仕組みのことをルーティングと呼ぶ。インターネットは、小規模なネットワークが細かい網の目のようになることで世界中を繋いでおり、ルーター同士が、接続されることでつながっている。データパケットをあて先のホストにきちんと届けるためには、各ルーターが正しい経路にデータを転送する必要がある。各ルーターは、自分の持つ経路制御表 (ルーティングテーブル) を参照して、データを転送する。ルーティングテーブルが異なっていれば、ルーターはデータを正しい方向に転送できず、データは目的地まで届かないことになる。

6.1.1 経路制御表 (ルーティングテーブル)

ルーティングテーブルとは IP パケットの中継装置（ルータ）や TCP/IP の通信ができる端末、コンピュータが持つ「次に IP パケットを送るべき相手」を指定したテーブルである。中継装置や端末は IP パケットを送る時は必ずこのテーブルを見て、IP パケットを送るべき相手を決定する。もし、IP パケットを送るべき最終目的地が同じイーサネットに接続されている場合は直接相手に IP パケットを送りますが、相手が異なるネットワークに接続されている時はルーティングテーブルを参照し相手に最も近いと思われる中継装置（ルータ）に IP パケットを送る。
ルーティングテーブルの作成方法にはスタティクルーティングとダイナミックルーティングと言う方法がある。

- 1. スタティクルーティング
  この方法では予め固定的にルートを設定する。ルートが固定的に設定されるので指定されたルートで障害が起きた場合は代替ルートに切り替わらず通信が途絶えてしまう。

- 2. ダイナミックルーティング
  この方法は他の中継装置（ルータ）から送られてくる情報（ルーティング情報）を使って定期的にルーティングテーブルを更新する方法である。この方法では予めルート情報を設定しておく必要は無いし、通信ルートに障害が起きた場合でも、その通信ルートを使って送られてくるルーティング情報そのものが途絶えるため、別のルートに自動的に切り換えられる。従って、幹線系ネットワークにおいて特別な理由が無い限り、ルーティングはダイナミックルーティングを使用するのが一般的となっている。

ダイナミックルーティングが一般的となっているため、複数回通信を行っていると、途中で経路が変わる場合もあるが、複数本の経路を意図的に用いて

通信することはできない。そこで、経路制御を利用して、複数本の経路を用意する。

6.2 経路制御（ソースルーティング）

ネットワーク・プロトコルは一般的に階層（レイヤ）という概念をもとに開発されていて、各階層ごとに役割が異なる。今回の目的である経路制御はネットワーク層で行われる。これは、ネットワーク上でのパケットの移動を制御する。先に説明したように、TCP/IP のルーティング経路は途中のルータが自動的に指定するようになっているが、ネットワーク層、TCP/IP ネットワークのルーティングテーブルに、パケットの通過経路を送信者が指定するのが、ソースルーティングである。途中のすべての経路を指定する「ストリクトソースルーティング」(strict source routing) と、いくつかの経路を通過することを指定して、それ以外は途中のルータに任せる「ルーズソースルーティング」(loose source routing) の 2 種類がある。図 6.1 のように、最短経路ではなく迂回させた経路を辿らせることによって、n 本の経路を確保する。赤の線が最適化された経路であり、緑や青は迂回させたルートとなっている。

図 6.2 のような情報をヘッダーにつけることで、経路制御は実現できる。

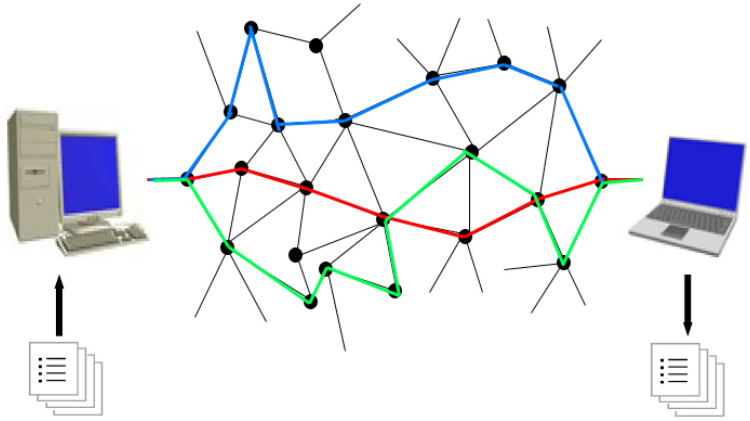


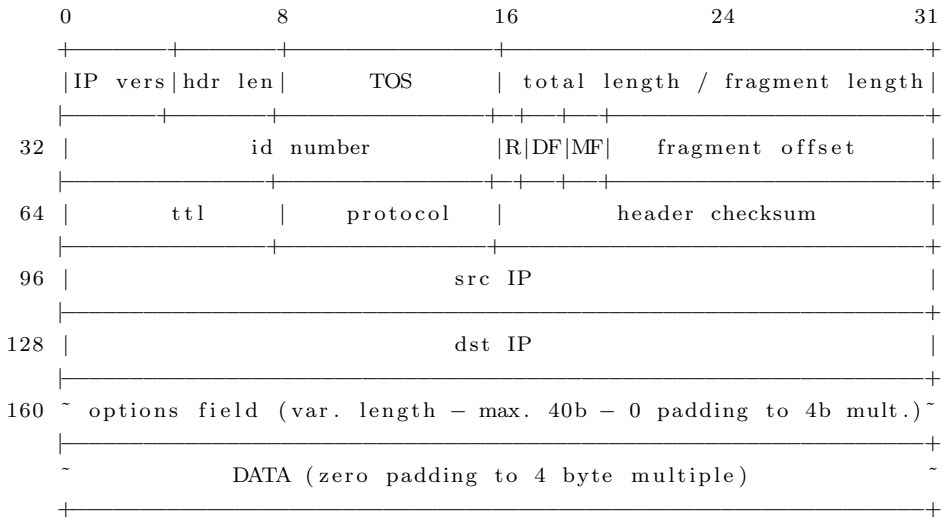
図 6.1　経路制御（ソースルーティングのイメージ）

39バイト

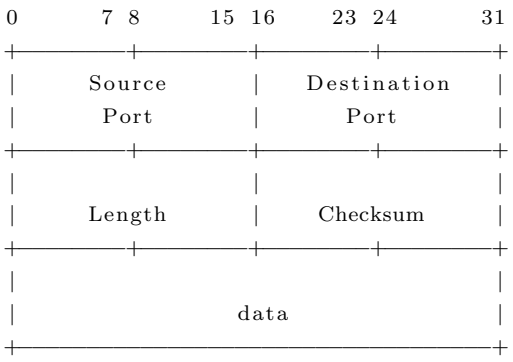
コード	データ長	ポインタ	IP アドレス＃1	IP アドレス＃2	IP アドレス＃3	・・・・・	IP アドレス＃9
1	1	1	4バイト	4バイト	4バイト	・・・・・	4バイト

図 6.2　始点経路制御指定の形式

6.3 ヘッダー



IP ヘッダー



UDP ヘッダー

ヘッダーはこのようになっており、IP ヘッダのデータの所に、UDP ヘッダが入り、そのデータ部分に図 6.2 のような、経路制御表を入れることで、経路制御が実現できる。

6.4 実験

・実験環境

dora14 が入ったマシン　2 台（実機）

・実験方法

この2台の間でパケットを任意のルートで通るかを実験した例えば、A→B→A→Bというルートを通って通信させるなど他のルーターなどをとしても実験したかったが、ファイヤーウォールなど、セキュリティーの関係でできなかった。

```
11:02:42.226116 IP pudding.ee.kanagawa-u.ac.jp.52564 > sattin.12344:
UDP, length 5
        0x0000:  0023 18ec 995a 001c c058 920c 0800 4800  .#...Z...X....H.
        0x0010:  002d 0000 4000 4011 b283 8548 58e7 8548  .-..@.@....HX..H
        0x0020:  58e4 0183 0b04 8548 58e4 8548 58e4 cd54  X......HX..HX..T
        0x0030:  3038 000d 6259 4845 4c4c 4f00                    08..bYHELLO.
```

上の数字の羅列はTCPDumpで経路上にあるマシンでパケットを見たものである。TCPDうｍｐとは、パケットを見ることができるソフトで、今回は

#tcpdump host 133.72.88.231 –XX

（133.72.88.231からのパケットのみを表示とし観察した）

8548 58e4 （133.72.88.228）から送信され、8548 58e7 （133.72.88.231）を経由し、8548 58e4 （133.72.88.228）で受信するように設定してあることがわかる。

第7章

経路制御（ソースルーティング）プログラム

7.1　受信プログラム

今回は、UDP 通信のプログラムを作成した。TCP を選択しなかったのは、パケットシーケンスチェックによる欠損パケット再送などのエラー訂正機能などを持っているため、複雑になる為である。
また、このプログラムは、エラー処理などを省き最低限のプログラムとなっている。

ソースコード 7.1　UDP 受信プログラム

```
1   #include <stdio.h>
2   #include <sys/types.h>
3   #include <sys/socket.h>
4   #include <netinet/in.h>
5
6    int
7   main()
8   {
9     int sock;
10  //ソケットを作る
11    struct sockaddr_in addr;
```

```
12
13   char buf[2048];
14
15    sock = socket(AF_INET, SOCK_DGRAM, 0);
16
17   addr.sin_family = AF_INET;
18   addr.sin_port = htons(12345);
19   //bindするIPアドレスとポートを設定する
20   addr.sin_addr.s_addr = INADDR_ANY;
21
22    bind(sock, (struct sockaddr *)&addr, sizeof(addr));
23   //ソケットに名前をつける（bindする）
24
25    memset(buf, 0, sizeof(buf));
26   recv(sock, buf, sizeof(buf), 0);
27  //データを受け取る
28
29   printf("%s\n", buf);
30
31    close(sock);
32
33    return 0;
34   }
```

7.2 送信プログラム

ソースコード 7.2 UDP送信プログラム

```
1   #include <stdio.h>
2   #include <unistd.h>
3   #include <sys/types.h>
4   #include <sys/socket.h>
5   #include <netinet/in.h>
6   #include <arpa/inet.h>
7   #include <memory.h>
8
9   #define IPOPT_NOP       1
10
11   int
12   main(int argc, char *argv[])
13   //配列の用意
14   {
15   int sock;
16   struct sockaddr_in addr;
```

```
17
18   int n;
19   if (argc != 2) {
20   fprintf(stderr, "Usage : %s dstipaddr\n", argv[0]);
21   return 1;
22   }
23
24  sock = socket(AF_INET, SOCK_DGRAM, 0);
25
26   addr.sin_family = AF_INET;
27   addr.sin_port = htons(12345);
28   //受信プログラムと同じポートを指定する
29   //inet_pton(AF_INET, argv[1], &addr.sin_addr.s_addr);
30   addr.sin_addr.s_addr = inet_addr("192.168.1.11");
31    //宛先のアドレスの宣言
32
33   typedef struct
34   {
35   unsigned char Nop;
36   unsigned char Code;
37   unsigned char Len;
38   unsigned char Offset;
39   unsigned long Addrs[2];
40   }LSR;
41
42   LSR SourceRoute;
43   memset(&SourceRoute,0,sizeof(LSR));
44
45   SourceRoute.Nop = IPOPT_NOP;
46   SourceRoute.Code = 0x89;
47   //今回は、SSRRの宣言だが、0x83とするとSLRRになる
48   SourceRoute.Len = 11;
49   SourceRoute.Offset = 4;
50   SourceRoute.Addrs[0] = inet_addr("192.168.1.10");
51   SourceRoute.Addrs[1] = inet_addr("192.168.1.11");
52  //通すIPアドレスの宣言  今回は192.168.1.11が終点アドレス
53
54   n = sendto(sock, "HELLO", 5, 0, (struct sockaddr *)&addr, sizeof(addr));
55   //送るメッセージの記述
56   if (n < 1) {
57   perror("sendto");
58   return 1;
59   }
60
61   close(sock);
62
63   return 0;
```

```
64    }
```

7.3 設定

ソースルーティングは禁止されている可能性があるので、以下の項目をチェックする。今回使用したFedora14の場合では、各ホストで、/etc/sysctl.confを以下のように修正する。

```
net.ipv4.conf.default.rp_filter=1
net.ipv4.conf.all.rp_filter=1
net.ipv4.ip_forward=1
net.ipv4.conf.all.accept_source_route = 1
```

第8章

結論

nチャンネル通信を実現するために、パケットの先頭にルーティングテーブルを書き込むプログラムを作成した。ソケットの扱いに慣れなかったため、実装するにはさまざまな課題が残ってしまったが、好きな経路を辿らせることができることは検証できた。

謝辞

本研究を行なうにあたり，終始熱心に御指導していただいた木下宏揚教授、鈴木一弘先生に心から感謝致します。良き研究生活を送らせていただいた木下研究室の方々に深く感謝致します.

参考文献

[1] DANNY DOLEV、CYNTHIA DWORK、ORLI WAARTS、MOTI YUNG ”Perfectly Secure Message Transmission”
Journal of the Association for Computing Machinery、Vol.40,No.1、pp.17-47(1993)

[2] K. Srinathan、Arvind Narayanan、C. Pandu Rangan ”Optimal Perfectly Secure Message Transmission”
CRYPTO 2004、LNCS 3152、pp.545-561(2004)

[3] Kaoru KUROSAWA、Kazuhiro SUZUKI、Members ”Almost Secure (1-Round,n-Channel) Message Transmission Scheme”
IEICE TRANS. FUNDAMENTALS、VOL.E92-A,NO.1(2009)

[4] 笠原正雄、佐竹賢治：”誤り訂正符号と暗号の基礎数理”、コロナ社 (2004)

[5] 澤田秀樹：”暗号理論と代数学”、海文堂 (1997)

[6] ”整数の合同” http://www2.cc.niigata-u.ac.jp/ takeuchi/tbasic/BackGround/-Cong.html

[7] 三谷政昭、佐藤伸一、ひのきいでろう：”マンガでわかる暗号”、オーム社 (2007)

[8] 黒澤馨、尾形わかは：”現代暗号の基礎数理”、コロナ社 (2004)

[9] 今井秀樹：”情報・符号・暗号の理論”、コロナ社 (2004)

[10] Douglas R. Stinson、櫻井幸一：”暗号理論の基礎”、共立出版株式会社 (1996)

[11] DOUGLAS R. STINSON：”CRYPTOGRAPHY THEORY AND PRACTICE THIRD EDITION”、Chapman & Hall/CRC(2006)

[12] HASAN MD. SAYEED、HOSAME ABU-AMARA ”Efficient Perfectly Secure Message Transmission in Synchronous Networks”
INFORMATION AND COMPUTATION 126、pp.53-61(1996)、ARTICLE NO.0033

[13] Saurabh Agarwal、Ronald Cramer、Robbert de Haan ”Asymptotically Optimal Two-Round Perfectly Secure Message Transmission”
CRYPTO 2006、LNCS 4117、pp.394-408(2006)

[14] Kaoru Kurosawa、Kazuhiro Suzuki ”Truly Efficient 2-Round Perfectly Secure Message Transmission Scheme”
Advances in Cryptology、EUROCRYPT 2008 LNCS 4965、pp.324-340(2008)

[15] ”as3crypto” http://code.google.com/p/as3crypto/

[16] . リチャード・スティーヴンス、篠田陽一：
”UNIX ネットワークプログラミング　第 2 版　Vol.1　ネットワーク API: ソケットと XTI”
株式会社ピアソン・エデュケーション（1999）

[17] . リチャード・スティーヴンス、橘康雄、井上尚司：”[新装版] 詳解 TCP/IP Vol.1 プロトコル”
株式会社ピアソン・エデュケーション（2000）

[18] . リチャード・スティーヴンス、徳田　英幸、戸辺義人：”[新装版] 詳解 TCP/IP　Vol.2 プロトコル”
株式会社ピアソン・エデュケーション（2002）

[19] きみち： ”基礎と実践 Linux ネットワークプログラミング”
ソフトバンククリエイティブ株式会社（2010）

[20] ichael J.Donahoo、Kenneth L. Calvert、小高知宏：”TCO/IP ソケットプログラミング　C 言語編”
オーム社（2003）

[21] 松公保：”基礎からわかる TCP/IP ネットワーク実験プログラミング　第 2 版”
オーム社（2004）

[22] 井康孝：”猫でもわかるネットワークプログラミング　第 2 版”
ソフトバンククリエイティブ株式会社（2006）

質疑応答