目 次

第1章　序論

1.1　研究の概要

近年、IoT(Internet of Things) 機器の導入が進んでいる。IoT とは、世の中にある様々な機器をインターネットに接続したり、相互に通信を行ったりすることにより、様々な情報をフィードバックする仕組みのことを指す。総務省による世界の IoT デバイス数の推移および予測によると、世界の IoT デバイス数は 2015 年では 165.6 億台、2017 年では 208.7 億台、2019 年では 253.5 億台となっている [1]。IoT 機器は家電や自動車のほか、温度・湿度センサーや IC タグ等、適用範囲は広く、機器の処理能力の高さは様々である。そのため、今後は処理能力が低い IoT 機器にも搭載可能なセキュリティ対策が不可欠となる。例えば、IoT 機器をインターネットに接続させて利用する際や、別の機器と無線通信による情報のやり取りを行う際には、暗号通信や暗号通信を行うための鍵配送を行う手法が求められる。

そこで本研究では、IoT 機器向けの暗号通信システムの実装に向けて、SAS-L2(Simple And Secure password authentication protocol,Light processing version, type 2) を研究対象とした。SAS-L2 とは、高知工科大学の清水明宏教授らが開発したワンタイムパスワード認証方式の 1 つである。SAS-L2 の特徴は、従来の SAS と比較し、被認証側の演算負荷が削減されていることである。そのため、処理能力の低い、温度センサや湿度センサ等の IoT 機器とそれらの機器を集約する装置の間で、ほぼ処理負荷なしに暗号通信の鍵配送実現が期待されている [2]。

また関連技術として、SAS-L2 と同時に開発されたのが SAS-L1 である。SAS-L1 は従来の SAS と比較して、認証側の演算負荷が削減されていることが特徴で、クラウドシステムやアプリケーション提供サービスシステム等のログイン時の認証への適用が想定されている [2]。SAS-L1 と SAS-L2 をまとめて SAS-L と呼ぶ。

SAS-L を用いた暗号通信システムの例として、図 1.1 に示す構成が考えらえれている。図 1.1 の想定するシステムでは、センサ等の複数の IoT 機器が取得したデータを集約装置に送信、集約装置からデータをクラウドサーバに送信し、クラウドでデータの分析・統計処理が行われる。この構成を想定した場合、IoT 機器-集約装置間の認証・鍵配送には SAS-L2 を、

集約装置-クラウドサーバ間の鍵配送では SAS-L1 を使用することで、各認証プロトコルの利点を有効活用した、演算負荷が軽量な暗号通信システムの実現が可能になると考えられる。

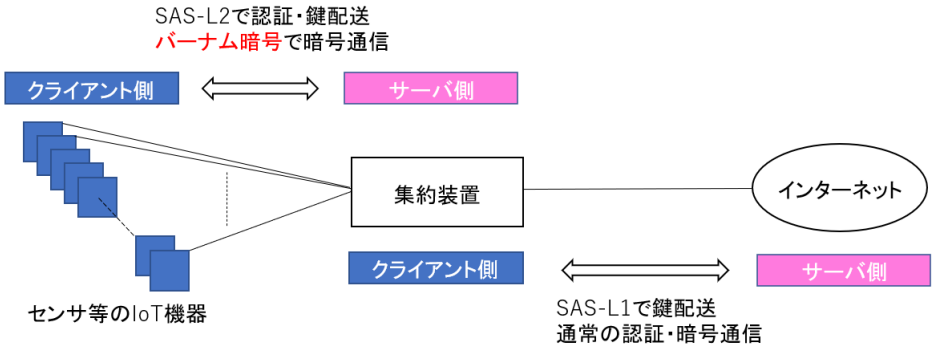

図 1.1　SAS-L による暗号通信 [2]

しかしながら、現段階では、SAS-L を実際の機器に搭載・動作させた例は存在せず、理論上の評価に留まっている。そのため、本研究の目的は、SAS-L2 を採用したプログラムを組込みデバイス上に実装することで、SAS-L2 に有意性があることを実際に確認することに加え、実装における SAS-L2 の問題点を明らかにすることである。そのために本研究の目標を、SAS-L2 によって鍵配送を行う暗号通信プログラムの実装、評価とする。比較対象となる従来の SAS として、本研究では SAS-2 を用いる。また、本研究における評価項目を CPU 時間と CPU 使用率、メモリ使用量と設定し、実装したプログラムを組込みボードの 1 つである Jetson nano に搭載して評価実験を行う。

本論文の構成は以下の通りである。第 1 章では、研究の背景および目的について述べる。第 2 章では、実装したプログラム内で用いた要素技術の原理について述べる。第 3 章では、SAS-2 および SAS-L2 の原理について述べる。第 4 章では、実装したシステムについて述べる。第 5 章では、実装したシステムを用いた評価実験の方法および結果を示した後、実験結果に基づいた考察を述べる。第 6 章では、本研究のまとめおよび今後の課題を述べる。

第2章　要素技術

本章では、実装したプログラム内で用いた要素技術の原理について述べる。

2.1　共通鍵暗号

共通鍵暗号とは、通信するデータに対して、暗号化や復号を行う際に同一の秘密鍵を用いる暗号のことを指す。秘密鍵とは暗号化アルゴリズムや復号アルゴリズムの動作を制御するためのデータのことである。以下、本稿では暗号化を施す対象となるデータのことを平文、平文に暗号化を施したデータのことを暗号文と呼ぶこととする。共通鍵暗号は鍵生成アルゴリズム KeyGen、暗号化アルゴリズム Enc、復号アルゴリズム Dec の組 (KeyGen,Enc,Dec) で構成されている (図 2.1)。

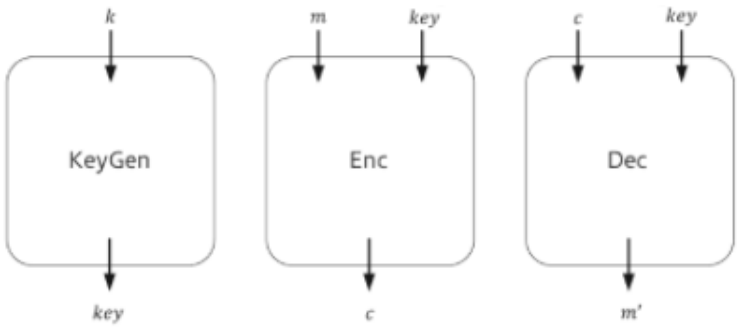

図 2.1　共通鍵暗号のアルゴリズム [3]

KeyGen はセキュリティパラメータ k を入力すると、秘密鍵 key を出力する。セキュリティパラメータとは、秘密鍵のサイズを決定する値である。Enc は平文 m と key を入力すると、暗号文 c を出力する。Dec は c と key を入力すると、復号結果である m' を出力する。元の平文と一致していれば、$m' = m$ となる。共通鍵暗号は、一般的には高速に暗号化・復号できるように設計されているため、大きなサイズの平文の暗号化に向いている。

共通鍵暗号は、暗号化する平文の単位によって、ストリーム暗号とブロック番号に分類されている。また、これらの分類に含まれないが、ストリーム暗号の原理に近いバーナム暗号が存在する。バーナム暗号については別項で述べる。

ストリーム暗号

ストリーム暗号とは、平文をビットやバイトなどの小さい単位で順次処理する暗号のことを指す。平文に対して、秘密鍵から生成した擬似乱数列と排他的論理和を適用させて暗号化することが可能である。暗号文に対して、同じ擬似乱数列との排他的論理和を再度適用させることで、平文を得ることが可能である。このタイプの暗号では、秘密鍵にあたるデータを暗号化処理に先んじて生成することが可能であること、送信するデータ (平文) に対してパディング処理を施す必要がないことにより、待ち時間が少なく高速に処理が行えるかつデータサイズが増加しないということがメリットとして挙げられる。しかしながら、ストリーム暗号では暗号化に擬似乱数を使うため、秘密鍵に大きく影響された値が出力されることがあったりする等、擬似乱数系列から秘密鍵の部分情報が推定できる危険性を考慮しなければならないことがデメリットとして挙げられる。

ブロック暗号

ブロック暗号とは、平文を一定の大きさ、すなわちブロック単位ごとに分割し、分割された平文に対して暗号化を施す共通鍵暗号の総称である。復号も暗号化の際に用いた秘密鍵を復号部に入力として与え、ブロック単位ごとに復号を行う。暗号化部および復号部の出力はブロック単位ごとに分割されたデータであるため、入力ブロックの順にデータを結合させることにより、暗号文、平文を得る。代表的なアルゴリズムに DES や AES などがあり、多くのブロック暗号では、暗号化部および復号部でそれぞれ同じ変換処理を繰り返すような構造となっている。

2.1.1　バーナム暗号

バーナム暗号とは、1917 年にバーナムによって考案された共通鍵暗号の一つである。バーナム暗号は 1949 年、シャノンによって理論的に解読不能であることが数学的に証明されている。

平文に対して、平文と同じ長さの乱数列 (秘密鍵) と、ビットごとに排他的論理和をとることによって暗号化が行われる。暗号文に対して、秘密鍵と、ビットごとに排他的論理和をとることによって復号が行われる。この時、暗号化および復号に用いる秘密鍵は、通信を行

う両者が事前に共有した乱数列であることが前提である。バーナム暗号の特徴は、秘密鍵が真性乱数から生成したビット列でなければならないことである。真性乱数とは、以下に示す 3 つの性質をすべて満たす乱数である。

無作為性

無作為性とは、統計的な偏りがない性質のことを指す。

予測不可能性

予測不可能性とは、過去の数列から次の数が予測できない性質のことを指す。つまり、乱数系列の任意の一部から、他のビットを効率的に推測できないことを意味している。

再現不可能性

再現不可能性とは、同じ数列を再現できない性質のことを指す。

真性乱数の生成には、放射線観測機による出力、周囲の温度や雑音の変化、マウスポインタの位置情報等の再現不可能な物理的な現象に対応する乱数を入力とすることで、同じ入力が得られないため、生成のたびに異なる乱数系列を出力する。

バーナム暗号は真性乱数から生成したビット列を使い捨ての秘密鍵として使用し、次回の通信では異なる秘密鍵を用いて行う。そのため、安全性が高い一方で、通信ごとに求められる秘密鍵生成のコストや秘密鍵の配送が課題として挙げられており、実用性が低く、限定的な状況でのみ使用されている [3]。

2.2　一方向性関数

関数 $f : \{0,1\}^* \to \{0,1\}^*$ が以下の性質を満たすとき、この関数 f を一方向性関数という。

効率的計算可能性

計算量が限定された多項式時間で実行できる確率的アルゴリズムを PPT(確率的多項式時間) アルゴリズムと呼び、一方向性関数においては、任意の入力 $x \in \{0,1\}^*$ に対し、$C(x) = f(x)$ となる PPT アルゴリズム $C : \{0,1\}^* \to \{0,1\}^*$ が存在する。

逆関数の困難性

任意の $c \in \mathbb{N}$ に対して、$k_0 \in \mathbb{N}$ が存在し、$k \geq k_0$ である任意の k に対して $\epsilon(k) < \frac{1}{k^c}$ を満たすとき、関数 $\epsilon : \mathbb{N} \to \mathbb{R}$ を無視可能関数と呼び、 一方向性関数においては、任

意の PPT アルゴリズム A と任意の $k \in \mathbb{N}$ に対して、無視可能関数 ϵ が存在し式 2.1 を満たす:

$$Pr[f(z) = y \,|\, x \leftarrow \{0,1\}^k, y = f(x), z \leftarrow A(1^k, y)] \leqq \epsilon(k). \tag{2.1}$$

一方向性関数の応用例としては、擬似乱数生成器や暗号学的ハッシュ関数などが挙げられる [4]。

2.3　ハッシュ関数

ハッシュ関数とは、任意の長さのデータを入力とし、そのメッセージを代表とする固定長の値を出力するデータ圧縮関数のことを指す。さらに、同じハッシュ関数かつ同じ入力データが与えられた場合は同じ出力データ (以下、ハッシュ値) が得られる。異なるデータを入力として与えた場合は、類似度にかかわらず、全く異なるハッシュ値を出力する。また、ハッシュ関数の標準的な安全性として以下の 3 つの性質が挙げられる。

一方向性 (原像計算困難性)

ハッシュ関数の出力値であるハッシュ値が与えられたとき、元のメッセージを求めることが困難な性質を指す。

第 2 原像計算困難性

あるメッセージとそのハッシュ値が与えられたとき、同一のハッシュ値になる別のメッセージを計算することが困難である性質のことを指す。

衝突困難性

同じハッシュ値となるような 2 つの異なる任意のメッセージを求めることが困難である性質を指す。ハッシュ値が指定されている状況における性質である第 2 原像計算困難性と比較すると、任意のハッシュ値に対して衝突が発生する場合を考えればよい衝突困難性の方が破られやすい。

これらの性質を満たすハッシュ関数のことを暗号学的ハッシュ関数と呼び、代表例として、SHA-256(Secure Hash Algorithm 256-bit) や MD5(Message Digest 5) が挙げられる。

次に、これらの性質を破るために総当たりによって入力を切り替えながら目標のハッシュ値を得ようとした場合、必要な計算回数の期待値は表 2.1 の通りである。一方向性では期待値が 2^{n-1} 程度の計算回数、すなわちハッシュ値空間の半分に相当するハッシュ値の計算を行うときの総当たり回数となる。第 2 原像計算困難性でも同様に 2^{n-1} 程度の計算回数、衝突

困難性では $2^{\frac{n}{2}}$ 回程度の計算回数である。衝突困難性は一方向性や第 2 原像計算困難性と比較して、少ない回数で破られるため、暗号学的ハッシュ関数に対して安全性を求める場合、まずは衝突困難性を満たす必要がある [3]。

表 2.1　手当たり次第計算する場合の平均的な計算回数 [3]

安全性の種類	破るために必要な計算回数
一方向性	2^{n-1} 回程度
第 2 原像計算困難性	2^{n-1} 回程度
衝突困難性	$2^{\frac{n}{2}}$ 回程度

2.3.1　SHA-256(Secure Hash Algorithm 256-bit)

SHA-256 は、2000 年に米国標準技術局により提案されたハッシュ関数であり、2002 年にハッシュ関数 SHA-1 と共に FIPS(連邦情報処理標準)180-2 として制定されている。

SHA-256 とは、最長 $2^{64} - 1$ bit までの任意の長さのデータから 256bit のハッシュ値を生成するハッシュ関数である。暗号学的ハッシュ関数として設計されているため、ハッシュ関数の標準的な安全性を満たしている。SHA-256 に類似して、ハッシュ値が 224bit の SHA-224、384bit の SHA-384、512bit の SHA-512 が存在しており、これらをまとめて SHA-2 と呼ばれる。

以下では、ハッシュ関数 SHA-256 のアルゴリズムを説明する。まず、説明で使用する定義と記法については次の通りである。

定義と記法

- $+$ とは、2^{32} を法とした算術加算を表す。
- $\oplus$ とは、 ビット毎の排他的論理和を表す。
- $\vee$ とは、 ビット毎の論理和を表す。
- $\wedge$ とは、 ビット毎の論理積を表す。
- $||$ とは、 ビットの連結を表す。
- $ROTR^y(x)$ とは、 x を右に y ビット巡回シフトすることを表す。
- $SHR^y(x)$ とは、 x を右に y ビットシフトすることを表す。

- $\bar{x}$ とは、 x のビット反転を表す。

次に、SHA-256 において用いられる論理関数は以下の通りである。

$$
\begin{aligned}
Ch(x,y,z) &= (x \wedge y) \vee (\bar{x} \wedge y) \\
Maj(x,y,z) &= (x \wedge y) \vee (y \wedge z) \vee (z \wedge x) \\
\Sigma_0(x) &= ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x) \\
\Sigma_1(x) &= ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x) \\
\sigma_0(x) &= ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x) \\
\sigma_1(x) &= ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)
\end{aligned}
$$

次に、SHA-256 の前処理にあたるパディング処理は、以下の手順で行われる。

1. 入力メッセージ M に対し、メッセージ長が 512 bit の倍数になるように、M の末尾に下記のデータを付加する。

$$M \parallel 1 \parallel 0^k \parallel l$$

ここで、l は M のメッセージ長のビット列 (64 bit)、k は $l + 1 + k \equiv 448 (mod\ 512)$ を満たす正の最小値である。この処理をパディングという。

2. パディングされたメッセージは、N 個の 512 bit 単位のブロック $M^{(i)}$ に分割される。

$$M = M^{(1)} \parallel M^{(2)} \parallel \cdots \parallel M^{(i)} \parallel \cdots \parallel M^{(N)}$$

ここで、各々の $M(i)$ は、16 個のワード

$$M^{(i)} = M_0^{(i)} \parallel M_1^{(i)} \parallel \cdots \parallel M_{15}^{(i)}$$

からなる。

3. 初期値として

$$
\begin{aligned}
H_0^{(0)} &= 0x6a09e667 \\
H_1^{(0)} &= 0xbb67ae85 \\
H_2^{(0)} &= 0x3c6ef372 \\
H_3^{(0)} &= 0xa54ff53a \\
H_4^{(0)} &= 0x510e527f \\
H_5^{(0)} &= 0x9b05688c \\
H_6^{(0)} &= 0x1f83d9ab \\
H_7^{(0)} &= 0x5be0cd19
\end{aligned}
$$

を設定する。

次に、ハッシュ計算として、N 個のメッセージブロック $M^{(i)}, \cdots, M^{(N)}$ の $M^{(i)}$ に対して、$1 \leq i \leq N$ の順に以下を実行する。

メッセージ拡張

次式で定義する SHA-256 メッセージ拡張関数を用いて拡張メッセージ W_t を計算する。

$$
\begin{aligned}
W_t &= M_t^{(i)} \quad (0 \leq t \leq 15) \\
W_t &= \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16} \quad (16 \leq t \leq 63)
\end{aligned}
$$

圧縮関数

1. 8 個のバッファ変数を $(i-1)$ 番目のハッシュ値 $H^{(i-1)}$ で初期化する。

$$
\begin{aligned}
a_0 &= H_0^{(i-1)} \\
b_0 &= H_1^{(i-1)} \\
b_0 &= H_2^{(i-1)} \\
d_0 &= H_3^{(i-1)} \\
e_0 &= H_4^{(i-1)} \\
f_0 &= H_5^{(i-1)} \\
g_0 &= H_6^{(i-1)} \\
h_0 &= H_7^{(i-1)}
\end{aligned}
$$

2. $0 \leq t \leq 63$ に対して以下の計算を繰り返す。

$$
\begin{aligned}
T_1 &= H_t + \Sigma_1(e_t) + Ch(e_t, f_t, g_t) + K_t + W_t \\
T_2 &= \Sigma_0(a_t) + Maj(a_t, b_t, c_t) \\
h_{t+1} &= g_t \\
g_{t+1} &= f_t \\
f_{t+1} &= e_t \\
e_{t+1} &= d_t + T_1 \\
d_{t+1} &= c_t \\
c_{t+1} &= b_t \\
b_{t+1} &= a_t \\
a_{t+1} &= T_1 + T_2
\end{aligned}
$$

ただし、K_t は 32 bit の定数である。また、ここでの 1 回の演算を、本稿では 1 ステップとする。ステップの構造を図 2.2 に示す。

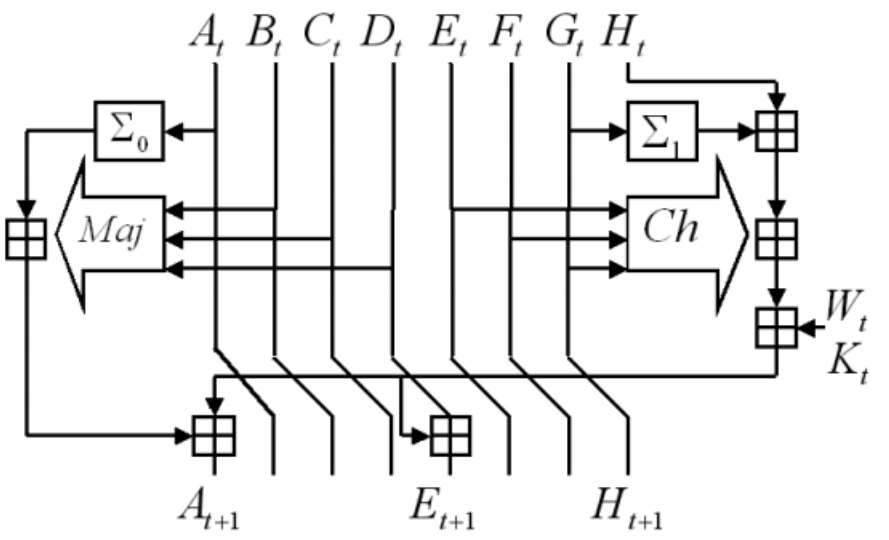

図 2.2　1 ステップの構造

3. i 番目の中間ハッシュ値を

$$H_0^{(i)} = H_0^{(i-1)} + a_{64}$$
$$H_1^{(i)} = H_1^{(i-1)} + b_{64}$$
$$H_2^{(i)} = H_2^{(i-1)} + c_{64}$$
$$H_3^{(i)} = H_3^{(i-1)} + d_{64}$$
$$H_4^{(i)} = H_4^{(i-1)} + e_{64}$$
$$H_5^{(i)} = H_5^{(i-1)} + f_{64}$$
$$H_6^{(i)} = H_6^{(i-1)} + g_{64}$$
$$H_7^{(i)} = H_7^{(i-1)} + h_{64}$$

で計算する。ここまでの 1 回の処理を、1block とする。ブロックの構造を図 2.3 に示す。上記手続きを N 回繰り返した最終的な 256 bit の値

$$H^{(N)} = H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)} \| H_5^{(N)} \| H_6^{(N)} \| H_7^{(N)}$$

がメッセージ M のハッシュ値である [5]。

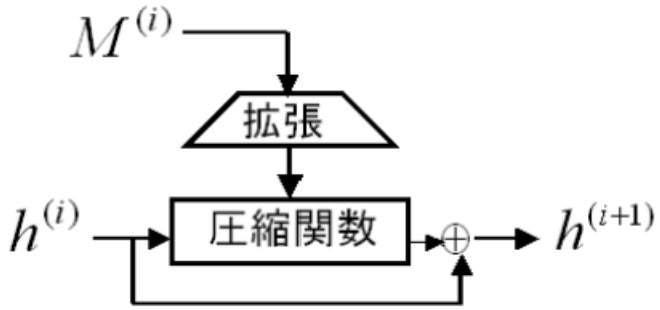

図 2.3　1block の構造

2.3.2　巡回冗長検査 (Cyclic Redundancy Check, CRC)

巡回冗長検査 は誤り検出符号の一種であり、任意の長さのデータを入力とし、その入力に対して一定の長さの区切りごとに固定長の検査ビットを付加する誤り検出方式である。1961 年に W. Wesley Peterson によって発表され、1975 年に CRC-32 と一般に呼ばれている IEEE 802.3 の CRC が定められた。CRC-32 は現在、イーサネットなどの各種通信や ZIP、PNG などの様々な場面で利用されている。

続いて、CRC 方式の概要を説明する。CRC 方式では、一定の生成多項式による除数の余りを検査用の冗長ビットとする方法である。受信した符号を表す多項式を $S(x)$ とするとき、$S(x)$ は 生成多項式 $G(x)$ で割り切れる符号であるため、割り切れない場合は受信データに誤りがあるとみなすことができる。情報ビットの多項式を $D(x)$、チェックビットの長さを k とするとき，符号の多項式 $S(x)$ は

$$S(x) = x^k \cdot D(x) + R(x)$$

で表せる。$R(x)$ は $x_k \cdot D(x)$ を $G(x)$ で割った余りでチェックビット (CRC) に対応する。生成多項式は唯一の標準規格があるわけではなく、選択する生成多項式 $G(x)$ によって誤り検出機能の高さや CRC 値の衝突のしづらさが変化する [6]。以下に巡回冗長検査の一つである CRC-32 および CRC-64 の生成多項式の例 [7] を示す。

(1) CRC-32(V.42 、MPEG-2、zlib)

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

(2) CRC-64(ECMA-182、ISO/IEC 13421)

$$\begin{aligned} &x^{64} + x^{62} + x^{57} + x^{55} + x^{54} + x^{53} + x^{52} + x^{47} + x^{46} + x^{45} + x^{40} + x^{39} + x^{38} + x^{37} + x^{35} \\ &+ x^{33} + x^{32} + x^{31} + x^{29} + x^{27} + x^{24} + x^{23} + x^{22} + x^{21} + x^{19} + x^{17} + x^{13} + x^{12} + x^{10} + x^9 \\ &+ x^7 + x^4 + x + 1 \end{aligned}$$

CRC 自体はデータ上の誤りを検出できるが、元データの復元はできない。ただし、他の誤り検出方式であるパリティチェックやチェックサムに比べて誤り検出精度が高く、バースト誤りにも強いという利点を持つ。

セキュリティ上において、一般的な CRC 方式におけるハッシュ値のビット数が小さく、同じ CRC 値になる複数のメッセージを容易に作成可能である (衝突困難性が破られる) ため、CRC 方式は暗号学的ハッシュ関数には含まれない。また、誤り検出には利用可能だが、CRC が変化しないように元のデータを改ざんすることが容易であることから、CRC 方式のみでは意図的なデータ改ざんを検出するといった用途には利用できない [6]。

2.4　暗号論的擬似乱数生成器

暗号論的擬似乱数生成器とは、暗号技術での利用に適した特性を持つ擬似乱数生成器の総称である。まず擬似乱数生成器とは、確定的な計算アルゴリズムにより乱数を生成する。初期値として、短いハードウェア乱数を入力し、長いハードウェア乱数系列に近い振舞う乱数系列を生成する。入力する乱数はシード (seed) と呼ばれる [8]。擬似乱数は、2.1.1 項に示した乱数の性質のうち、無作為性のみを持つ乱数、あるいは、無作為性と予測不可能性を満たす擬似乱数に分けられる。暗号技術で用いる乱数は、暗号学的に安全な乱数でなければならず、少なくとも予測不可能性を満たす擬似乱数である必要がある [3]。

暗号論的擬似乱数生成器をコンピュータで扱う方法の一つとして、カーネル内臓の乱数生成器を使う方法がある。例えば、Linux は/dev/random や/dev/urandom という乱数生成デバイス、Windows は `CryptGenRandom()` という専用の関数が存在する。本稿では、実装したプログラムの乱数生成として採用した/dev/urandom について述べる。

2.4.1　/dev/urandom

/dev/urandom は UNIX 系のカーネルに内蔵されている擬似乱数生成デバイスである。様々なハードウェアやデバイスドライバ等から推測困難な情報を得て、エントロピープールに格納される。/dev/urandom が読み込まれると、エントロピープール内のデータをハッシュ化し、乱数を生成する。ここで、エントロピープール内のデータが不足している場合は、データを再利用して乱数を生成する。/dev/urandom に関連して、/dev/random も同じ UNIX 系カーネル内臓の乱数生成器である。/dev/random との違いは乱数生成時にエントロピープール内のデータを全て消費した場合、乱数生成に必要な量のデータが蓄積するまでブロッキングされることである [3]。

第3章　SAS認証方式(Simple And Secure password authentication protocol)

本章では、鍵配送方式として利用可能なワンタイムパスワード認証方式 SAS の中でも本研究で取り扱う SAS-2 と SAS-L2 [9] のアルゴリズムについて述べ、両者の理論上の計算量を示す。これらの認証方式では、認証者側と被認証者側で共有される認証情報を共通鍵暗号方式による暗号通信の秘密鍵として利用することで暗号通信が可能となる。また、認証情報は認証を行うたびに変更されるため、秘密鍵の情報漏洩が発生してもその後の通信内容が盗まれ続けることがないという利点がある。

3.1　SAS-2

SAS-2 はインターネットでの利用に適したワンタイムパスワード認証方式 S/Key の速度性能を改善する方式として、2002 年に高知工科大学の清水明宏教授によって開発された。インターネットで利用されるワンタイムパスワード認証方式に S/Key があり、この方式では、一方向性関数を何度も繰り返し適用させて認証情報の作成を行う。一方で、SAS-2 における認証情報の作成では、 一方向性関数のほか、排他的論理和、加算をそれぞれ数回行うのみで実現可能である。特に、ワンタイムパスワード認証における演算負荷に大きく関わる一方向性関数の適用回数が削減されていることが大きな特徴である。SAS-2 では主に初回認証の前に行われる登録フェーズと、認証フェーズに分かれており、以降の項で示す流れで認証者-被認証者間の相互認証を実現する。

3.1.1　定義と記法

SAS-2 認証方式において使用する定義と記法については次の通りである。

- User とは、 被認証者を表す。
- Server とは、 User を認証する認証者を表す。
- ID とは、 User の識別子を表す。
- S とは、 User のパスワードを表す。
- H(s) とは、 s に対して一方向性関数を 1 度適用して得た演算結果を表す。
- N_i とは、 i 回目の認証時に生成される乱数を表す。
- + とは、 ビット毎の算術加算を表す。
- $\oplus$ とは、 ビット毎の排他的論理和を表す。

3.1.2　登録フェーズ

SAS-2 における登録フェーズでは, User 側で初期情報を生成し、 安全なルートを経由して Server に送られる. 図 3.1 に SAS-2 の登録フェーズのフローチャートを示す。

1. User は自身の識別子 ID, パスワード S を入力する。
2. User は 1 回目の認証用の乱数 N_1 を生成し保存する。
3. User は入力した S, 生成した N_1 を用いて 1 回目認証用の認証情報 $A_1 = H(S \oplus N_1)$ を演算する。
4. User は安全なルートを経由して A_1 を Server へ送信する。
5. Server は受け取った ID、A_1 を保存する。

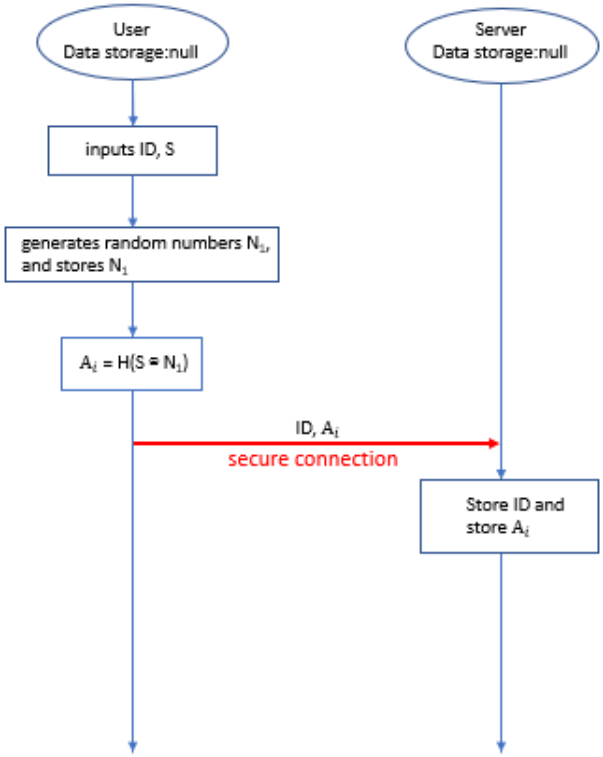

図 3.1　SAS-2 の登録フェーズ

3.1.3　認証フェーズ

SAS-2 における認証フェーズでは, まずは User から送られてきた認証情報を用いて、Server が今回認証情報と一致するかどうかを検証することで User の認証を行い、次に Server から送られてきた認証情報を用いて、User が今回認証情報と一致するかどうかを検証することで Server の認証を行う。これにより、User-Server 間の相互認証が可能となる。 図 3.2 に i 回目の認証における SAS-2 認証フェーズのフローチャートを示す。

1. User は自身のパスワード S を入力する。
2. User は入力したパスワードと保存されている乱数 N_i から i 回目認証用の認証情報 $A_i = H(S \oplus N_i)$ を演算する。
3. User は次回認証用の乱数 N_{i+1} を生成し保存する。
4. User は乱数 N_{i+1} から 次回認証用の認証情報 $A_{i+1} = H(S \oplus N_{i+1})$、$B = H(A_{i+1})$、

$\alpha = A_{i+1} \oplus (H(A_{i+1}) + A_i)$、$\beta = A_{i+1} \oplus A_i$ を演算する。

5. User は α, β を Server へ送信する. この時使用するネットワークはインターネットなどの安全でないルートを経由しても問題はない。

6. Server は受信した α, β と保存されている A_i を用いて $C = \beta \oplus A_i$、$D = \alpha \oplus (C + A_i)$、$E = H(D)$ を演算する。D の演算を行う際に桁あふれを起こした場合はその値を切り捨てるものとする。

7. Server は C と E を比較し、一致すれば User の認証が成功、以下の処理が実行される。不一致ならば User の認証は不成立となり、以下の処理は実行されない。

8. Server は保存されている A_i の代わりに D を次回の認証情報として保存する。

9. Server は $\gamma = H(C)$ を演算する。

10. Server は γ をインターネットなどを通して User へ送信する。

11. User は $E = H(B)$ を演算する。

12. User は E と受信した γ を比較し、一致すれば Server の認証が成功となり User-Server 間の相互認証が成功したこととなる. 不一致ならば認証は不成立となる。

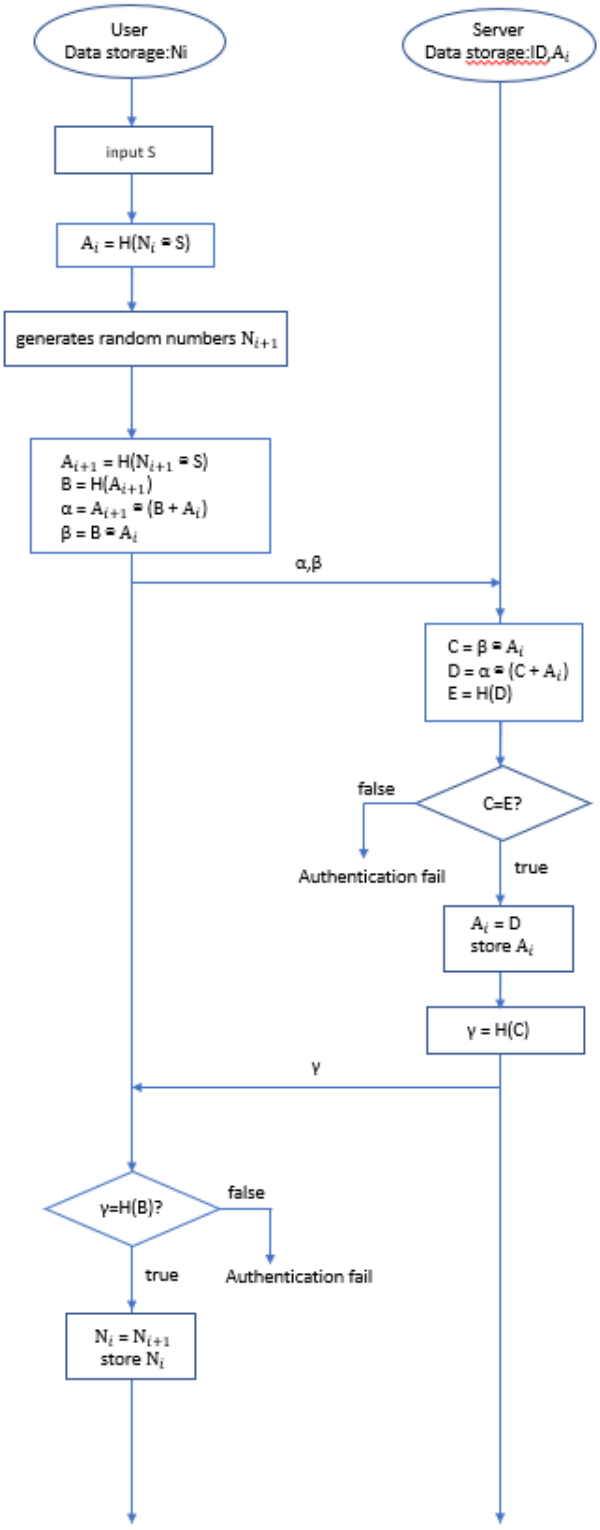

図 3.2　SAS-2 の認証フェーズ

3.2　SAS-L2

SAS-L2 は前節で述べた従来の SAS 認証方式である SAS-2 における被認証者側の演算負荷を改善する方式として 2018 年に清水明宏教授によって考案された。SAS-L2 の認証情報の作成では、被認証者側における一方向性関数の適用回数は 0 回となり、排他的論理和、加算は数回行うのみで実現可能であることが大きな特徴である。SAS-L2 は SAS-2 と同様に、初回認証の前に行われる登録フェーズと、認証フェーズに分かれているが、SAS-2 は一方向性変換関係の検証を認証の論拠にしているのに対して、SAS-L2 は今回認証情報と次回認証情報の演算が同一になるかどうかを検証することで認証を行う。SAS-L2 のアルゴリズムの詳細を次に示す。

3.2.1　定義と記法

SAS-2 認証方式において使用する定義と記法については以下の通りである。

- User とは、被認証者を表す。
- Server とは、 User を認証する認証者を表す。
- S とは、 User の識別子を表す。
- H(s) とは、s に対して一方向性関数を 1 度適用し、得た演算結果を表す。
- N_i とは、i 回目の認証時に生成される乱数を表す。
- M_i とは、 i 回目の認証時に生成されるマスク値を表す。
- + とは、ビット毎の算術加算を表す。
- $\oplus$ とは、 ビット毎の排他的論理和である。

3.2.2　登録フェーズ

SAS-L2 における登録フェーズでは、 Server 側で初期情報を生成し、 安全なルートを経由し、User に送られる。 図 3.3 に SAS-L2 の登録フェーズのフローチャートを示す。

1. Server は User 固有の識別子 S を入力する。
2. Server は 初回認証用の乱数 N_1 、マスク値 M_1 を生成し保存する。

3. Server は入力された S、生成された N_1 を用いて 初回認証用の認証情報 $A_1 = H(S \oplus N_1)$ を演算し、保存する。

4. Server は安全なルートを経由し、A_1、M_1 を User へ送信する。

5. User は受け取った A_1、M_1 を保存する。

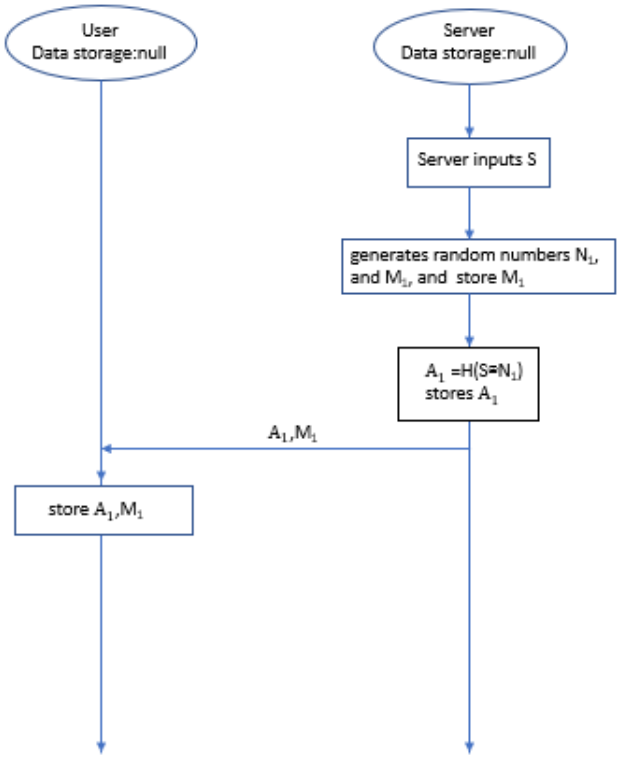

図 3.3　SAS-L2 の登録フェーズ

3.2.3　認証フェーズ

SAS-L2 における認証フェーズでは, まずは User から送られてきた認証情報を用いて、Server が今回認証情報と一致するかどうかを検証することで User の認証を行い、次に Server から送られてきた認証情報を用いて、User が今回認証情報と一致するかどうかを検証することで Server の認証を行う。これにより、User-Server 間の相互認証が可能となる。 図 3.4 に i 回目の認証における SAS-L2 認証フェーズのフローチャートを示す。

1. Server は User 固有の識別子　S を入力する。

2. Server は次回認証用の乱数 N_{i+1} を生成する。
3. Server は乱数 N_{i+1} から次回認証用の認証情報 $A_{i+1} = H(S \oplus N_{i+1})$、$\alpha = A_i \oplus A_{i+1} \oplus M_i$、$\beta = A_i + A_{i+1}$ を演算する。
4. Server は α を User へ送信する。以降、認証情報送信に使用するネットワークはインターネットなどの安全でないルートであっても問題はない。
5. User は受信した α と保存された A_i を用いて $B = \alpha \oplus A_i \oplus M_i$、$C = A_i + B$ を演算する。
6. User は C を Server へ送信する。
7. Server は受信した C と 先ほど演算した β を比較し、 一致すれば認証が成功、 以下の処理が実行される。不一致ならば認証は不成立となり以下の処理は実行されない。
8. Server は $M_{i+1} = A_i + M_i$、$\gamma = A_i \oplus M_{i+1}$ を演算する。M_{i+1} の演算を行う際に桁あふれを起こした場合はその値を切り捨てるものとする。
9. Server は保存されている A_i の代わりに A_{i+1} を新しい認証情報、M_i の代わりに M_{i+1} を新しいマスク値として保存する。
10. Server は γ を User へ送信する。
11. User は $M_{i+1} = A_i + M_i$、 $D = A_i \oplus M_{i+1}$ を演算し、D と受信した γ を比較する. 一致すれば Server の認証が成功となり User-Server 間の相互認証が成功したこととなる。不一致ならば認証は不成立となり以下の処理は実行されない。
12. User は保存されている A_i の代わりに B を新しい認証情報、M_i の代わりに M_{i+1} を保存する。

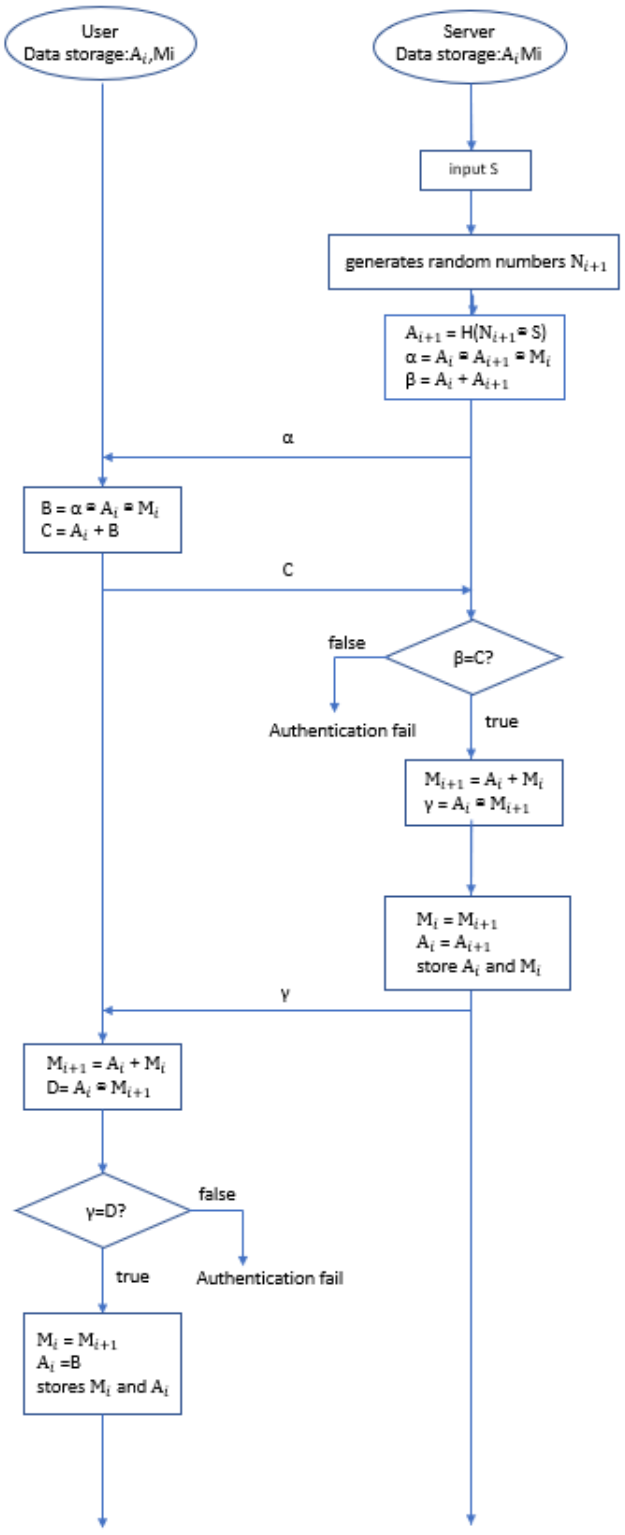

図 3.4　SAS-L2 の認証フェーズ

3.3　演算負荷の比較

SAS-2 および SAS-L2 では、認証情報の作成のために、一方向性関数、排他的論理和、加算のみの演算で実現しているため、演算処理にかかる負荷はそれらの演算の適用回数で比較し評価することが可能である。SAS-および SAS-L2 の演算負荷の比較表を表 3.1 に示す。

被認証者側において、一方向性関数の適用回数は、SAS-2 で 4 回適用される一方で、SAS-L2 は 0 回となっている。また、認証者側も同様に一方向性関数の適用回数が SAS-L2 の方が少なく、理論上の演算負荷については、SAS-L2 の方が軽量である (表 3.1)。

例えば、SHA-256 を用いる場合の演算負荷は以下のように評価できる。SHA-256 はハッシュ値の演算にあたって、メッセージ拡張のために、32bit の加算を 3 回、32bit の排他的論理和を 4 回、分割されたビットデータに対して右方向への 1bit ローテーションを 61 回、1bit の右シフトを 13 回、これを 48 回繰り返している。また、圧縮関数においては、32bit の加算を 7 回、32bit の排他的論理和を 4 回、32bit の論理和を 3 回、32bit の論理積を 5 回、分割されたビットデータに対して右方向への 1 ビットローテーションを 79 回、ビット反転を 1 回、これを 64 回繰り返している。後処理である出力値の生成の際に加算が 8 回行われる。

ビットローテーションは排他的論理和の処理負荷とほぼ等しいこと [2]、32bit 同士の演算においても 64bit のレジスタを用いると考えれば、これらの処理を集計すると、加算 600 回、排他的論理和 8432 回、右シフト 624 回、論理和 192 回、論理積 320 回、ビット反転 64 回に相当するため、加算や排他的論理和を数回程度行う場合の処理負荷は、一方向性関数適用による処理負荷と比較し、ほぼ無視できる。このことから、SAS 認証方式における処理負荷においては、一方向性関数の適用回数を比較することで評価することが可能である。

表 3.1　演算負荷の比較表

	被認証者側			**認証者側**		
	一方向性関数	XOR	加算	一方向性関数	XOR	加算
SAS-2	4	4	1	2	2	1
SAS-L2	0	3	2	1	4	2

第4章　暗号通信システムの実装

本章では、評価実験を見越して実装した暗号通信システムについて述べる。4.1 節でシステムの開発環境について示す。そして、4.2 節でシステム構成について述べ、4.3 節で評価実験を行うためにシステムに追加した特別な処理内容について述べる。最後に、4.4 節でプログラムの実行方法について述べる。

4.1　開発環境

暗号通信システムの実装に用いた開発環境を表 4.1 に示す。

表 4.1　開発環境

CPU	Intel(R)Core(TM) i7-7700 CPU @ 3.60GHz 3.60GHz
コア数	4
メモリ	16GB
SSD	512GB
ホスト OS	Windows 10 Pro
ゲスト OS	Debian 10.2.0

4.2　システム構成

本研究では、認証情報が 32bit、64bit、256bit の 3 パターンを想定し、複数の測定プログラムを実装した。実装した暗号通信システムは前処理、初回登録、認証、暗号通信の 4 部構成となっている (図 4.1)。

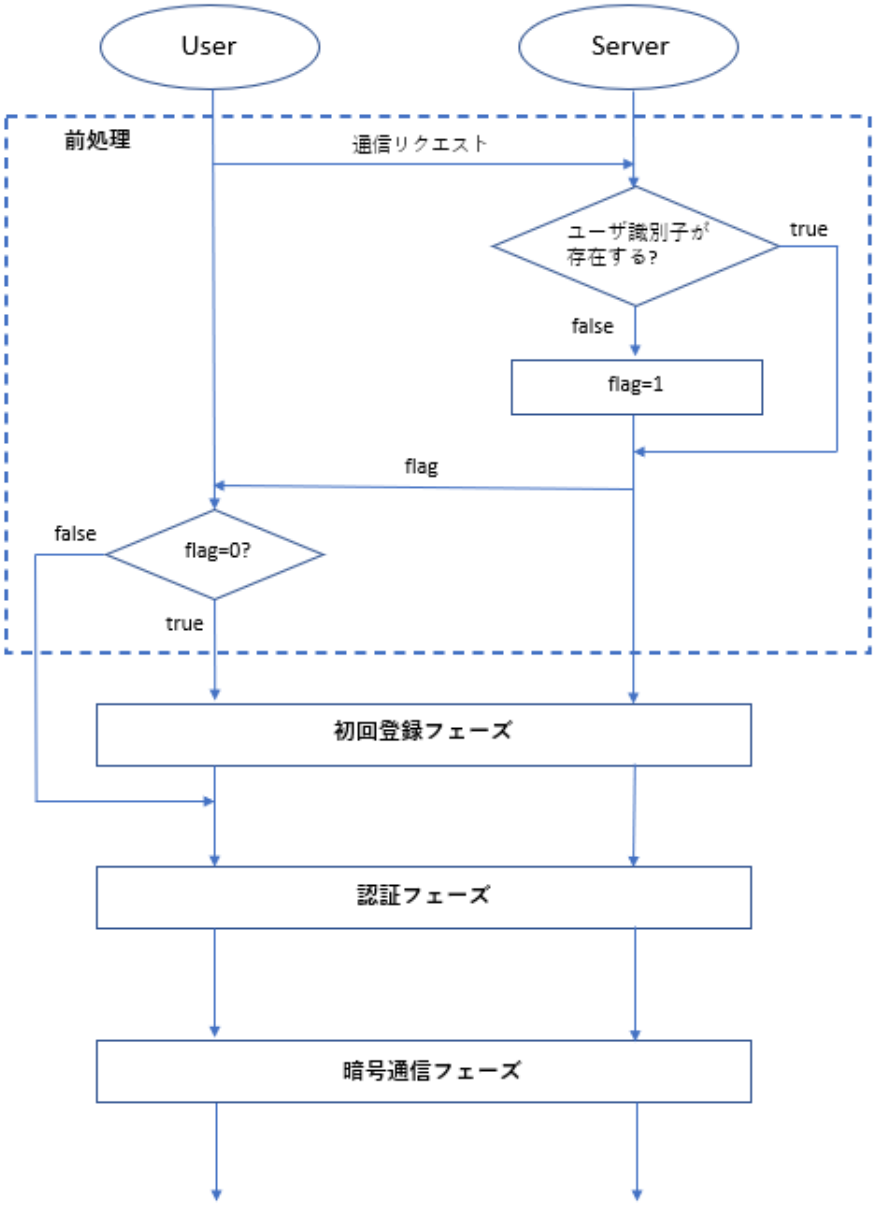

図 4.1　実装した暗号通信システムの構成

前処理では、認証者側が被認証者側からの暗号通信リクエストを受け取り、通信相手がすでに初回登録済であるかを調べ、フラグを被認証者側に送信する。通信相手となる被認証者が初回登録済であるかどうかは、事前に認証者側で管理しているユーザデータファイルを参照し、被認証者の IP アドレスに対応するユーザ識別子が存在するか否かで判断している。被認証者側は受け取ったフラグ内容によって、初回登録フェーズもしくは認証フェーズの処理に移行する。

初回登録フェーズおよび認証フェーズでは、3 章で示したアルゴリズムに従って処理を

行う。

SAS のアルゴリズムにおいて、一方向性関数の出力データと認証情報の排他的論理和をとる必要がある。そのため、情報損失を考慮し、認証情報のデータ長とハッシュ関数のハッシュ値を統一させることが望ましいが、32bit もしくは 64bit のハッシュ値を出力する暗号学的ハッシュ関数は存在しない。本研究では、一方向性関数の代替として、認証情報が 32bit の場合は CRC-32 を、64bit の場合は CRC-64 を用いた。認証情報が 256bit の場合は暗号学的ハッシュ関数の SHA-256 を用いた。

乱数生成には、暗号論的擬似乱数生成器の一つである、/dev/urandom を用いた。/dev/urandom を使用する際は、open システムコールで開き、乱数を取得する。

本研究で実装したプログラムで扱う認証情報は、特定のバイナリファイルに保存され、認証の度に更新される。

暗号通信フェーズでは、認証フェーズを通して共有された認証情報を秘密鍵としたバーナム暗号を用いて行われる。被認証者側で事前に特定のテキストファイルに保存された平文を秘密鍵によって暗号化し、認証者側へ送信される。認証者側は受け取った暗号文を秘密鍵によって復号し、その結果を特定のテキストファイルに保存する。

4.3　評価実験のための追加処理

本研究では、評価実験で得られる結果の精度を高めるため、本来の運用では不必要な処理をプログラムに追加している。

まずは、測定対象となる処理をループさせ、コマンドライン引数から動的にそのループ回数を設定できる仕様である。この追加処理は、CPU 時間計測時の出力値の精度を考慮している。評価項目の一つである CPU 時間の計測では、clock 関数を用いており、ミリ秒単位の CPU 時間が出力される。実装したプログラムにおいて、1 回の処理にかかる CPU 時間は短く、出力値の有効桁数が制限されるため、同じ処理を複数回ループさせて clock 関数で表現できる値の範囲に調整している。

次に、データの送受信に用いるソケットバッファのサイズを 100KB に設定した。記述方法は Listing4.1 に示す。この追加処理は、データ通信にかかる CPU 時間の精度向上が目的である。評価実験では、データ通信を含む処理を複数回ループさせるため、ソケットバッファのサイズが小さければ、バッファを全て消費する恐れがある。バッファを全て消費した場合、バッファの空きが発生するまで一時的にデータ送受信関数の処理がブロッキングされる。つまり、ソケットバッファのサイズがボトルネックとなり、データ通信にかかる CPU 時間が

SAS のアルゴリズムとは関係ない部分で長くなる現象を回避する必要がある。

Listing 4.1　ソケットバッファサイズの変更を行うコードの追加 [10]

```
1 int rcvBufferSize=100*1024;
2 if(setsockopt(sock, SOL_SOCKET, SO_RCVBUF, &rcvBufferSize,
3     sizeof(rcvBufferSize))<0){
4       perror("setsockopt() failed\n");
5     }
6 }
```

4.4　実行方法

認証情報が 32bit もしくは 64bit の場合のコンパイル方法と実行方法は次の通りである。

認証者側

コンパイル方法は次の通りである。ここでは、ソースファイル名を sasl2server.c、実行ファイル名を server とする。

```
% gcc sasl2server -o server
```

次に、実行方法は次の通りである。コマンドライン引数として、サーバのタイムアウト時間 (ex.1000s)、ポート番号 (ex.7777) を設定する。

```
% server 1000 7777
```

被認証者側

コンパイル方法は次の通りである。ここでは、ソースファイル名を sasl2client.c、実行ファイル名を client とする。

```
% gcc sasl2client.c -o client
```

次に、実行方法は次の通りである。コマンドライン引数として、接続先の IP アドレス (ex.192.168.26.40)、ポート番号 (ex.7777)、認証フェーズの繰り返しの回数 (ex.1000 回) を入力する。このとき、ポート番号はサーバプログラムにおいて開かれているポートのものである必要がある。

```
% client 192.168.26.40 7777 1000
```

認証情報が 256bit の場合のコンパイル方法と実行方法は次の通りである。

認証者側

コンパイル方法は次の通りである。ここでは、ソースファイル名を sasl2server.c、実行ファイル名を server とする。openSSL の sha.h をインクルードするため、リンクオプションとして-lcrypto を追加する。

```
% gcc sasl2server.c -o server -lcrypto
```

次に、実行方法は次の通りである。コマンドライン引数として、サーバのタイムアウト時間 (ex.1000s)、ポート番号 (ex.7777) を入力する。

```
% server 1000 7777
```

被認証者側

コンパイル方法は次の通りである。ここでは、ソースファイル名を sas2client.c、実行ファイル名を client とする。次に示す例ではリンクオプション-lcrypto が追加されているが、SAS-L2 の被認証者側プログラムでは追加する必要はない。

```
% gcc sas2client.c -o client -lcrypto
```

次に、実行方法は次の通りである。コマンドライン引数として、接続先の IP アドレス (ex.192.168.26.40)、ポート番号 (ex.7777)、認証フェーズの繰り返しの回数 (ex.1000 回) を入力する。このとき、ポート番号はサーバプログラムにおいて開かれているポートのものである必要がある。

```
% client 192.168.26.40 7777 1000
```

第5章　評価実験

本章では、評価実験について述べ、実験結果を示す。5.1 節では、評価実験環境を示す。5.2 節では、使用コマンドやパラメータ設定などの評価実験の実施に関する内容について述べる。5.3 節では、実験結果を示す。最後に、5.4 節では、実験結果に基づいた評価・考察を述べる。

5.1　評価実験環境

評価実験では、NVIDIA の組込み AI ボードである、Jetson Nano を使用した。Jetson Nano のスペック表を表 5.1 に示す。

表 5.1　Jetson Nano のスペック表

CPU	ARM Cortex-A57 1.43GHz
コア数	4
メモリ	4GB
GPU	Maxwell 920MHz
ストレージ	16GB
OS	Ubuntu 18.04.5 LTS

評価実験では、認証者側および被認証側にあたる機器を Jetson Nano とし、両機器をローカルに接続した状態で、認証情報の送受信を行った。本研究では、ノイズによる同期問題に関して考慮しないため、各機器をイーサネットで接続した。

5.2　実験方法

評価実験では、SAS-2 および SAS-L2 の認証フェーズを測定対象とした。ちなみに、初回登録フェーズは初回認証の前に 1 回のみ処理が実行されるということ、バーナム暗号による

暗号通信部における処理は SAS-2 と SAS-L2 で共通であるということから、運用時において、その部分にあたる処理負荷の差が大きく影響しないものと判断し、計測範囲から除外した。

SAS-L2 によって、処理負荷が削減できることを確認する必要があるため、評価項目を CPU 時間とリソース使用量 (CPU 使用率、メモリ使用量) とし、実装したプログラムを Jetson nano に搭載して評価項目の測定を行った。以下、各実験における実験方法の詳細を述べる。

5.2.1　実験 1 : SAS-L2 にかかる CPU 時間の計測

実験 1 における計測項目として、認証部全体の CPU 時間の他、演算、入出力、通信といった、認証フェーズの処理を構成する各機能のみをそれぞれ実行した場合の CPU 時間も含めた。アルゴリズムを SAS-2 から SAS-L2 に置き換えることで、各機能 (演算、入出力、通信) の処理負荷がどのように変化するかを検証することが目的である。

CPU 時間を測るにあたって、ソースコード内で clock 関数を使用した。clock 関数を通して認証フェーズの処理にかかる CPU クロック数を計測し、その値を CLOCKS_PER_SEC(=1000000) で除算することにより CPU 時間を求めた。このとき、CPU 時間にあたる値の有効桁数を確保するため、認証部を 1000 回ループさせた際の CPU 時間を求めた。同様の測定を 5 回行い、得られた CPU 時間の平均を実験結果の値とした。

5.2.2　実験 2 : SAS-L2 に必要なリソース使用量の調査

認証部の処理を行う際のリソース使用量として、CPU 使用率およびメモリ使用量の計測を行った。比較対象は実験 1 と同様に SAS-2 とする。まず、CPU 使用率の測定にあたり、次の Linux コマンドを用いた。被認証者側のプログラムを処理するプロセスに割り振られたプロセス番号が 3300 である場合の入力例を示している。

```
% pidstat -p 3300 1
```

計測項目として、認証部すべての処理を対象とする全体の CPU 使用率の他、認証情報の演算など、主に演算処理を対象としたアプリケーションによる CPU 使用率、データ通信処理やファイルへの入出力処理を対象としたカーネルによる CPU 使用率を含めた。これらの項目に対して、認証部の処理をループさせながらプログラムを 1 分間動作させ、その間に出力される CPU 使用率の平均値を求め、実験結果の値とした。

次に、メモリ使用量の測定にあたり、次の Linux コマンドを用いた。被認証者側のプログラムを処理するプロセスに割り振られたプロセス番号が 3300 である場合の入力例を示している。

```
% cat /proc/3300/status | grep -e VmHWM
```

対象のプロセスが使用した物理メモリサイズのピーク値 VmHWM を計測した。5.2 節の冒頭で、測定対象が認証フェーズであると述べたが、使用コマンドの仕様上、メモリ使用量の測定時のみプロセス全体を測定対象とした。つまり、認証フェーズの他、通信確立時や暗号通信フェーズで必要なメモリサイズを含めた値を実験結果の値とした。

また、メモリ使用量については、SAS のアルゴリズムには記載されていない要素、つまり、変数の型やインクルードしたヘッダファイルなどによって結果が大きく変化する。そのため、メモリ使用量に関する計測結果は参考用として付録 A に示すものとする。また、考察では、SAS-2 および SAS-L2 のアルゴリズムにおいて必要となるデータのみを取り上げ、必要メモリ量を比較した。

5.3　実験結果

5.3.1　実験 1 : SAS-L2 にかかる CPU 時間の計測

まず、認証部の処理を 1000 回行うためにかかった CPU 時間の計測結果を表 5.2 に示す。

表 5.2　各認証方式における認証部 1000 回分の CPU 時間

	32bit	**64bit**	**256bit**
SAS-2	1.426685	1.382164	1.226761
SAS-L2	1.293885	1.191140	1.107960

次に、各機能に限定した CPU 時間の計測結果を示す。表 5.3 は、認証情報のデータの長さを 32bit、ハッシュ関数として CRC-32 を使用した場合の計測結果である。

表 5.3　各認証方式における認証部の各機能 1000 回分の CPU 時間 (32bit, 単位 s)

	通信のみ	演算のみ	入出力のみ
SAS-2	0.353052	0.016580	0.186832
SAS-L2	0.582719	0.000040	0.193532

次に、表 5.4 は、認証情報のデータの長さを 64bit、ハッシュ関数として CRC-64 を使用した場合の計測結果である。

表 5.4　各認証方式における認証部の各機能 1000 回分の CPU 時間 (64bit, 単位 s)

	通信のみ	演算のみ	入出力のみ
SAS-2	0.354409	0.019744	0.183314
SAS-L2	0.568055	0.000020	0.154190

次に、表 5.5 は、認証情報のデータの長さを 256bit、ハッシュ関数として SHA-256 を使用した場合の計測結果である。

表 5.5　各認証方式における認証部の各機能 1000 回分の CPU 時間 (256bit, 単位 s)

	通信のみ	演算のみ	入出力のみ
SAS-2	0.365669	0.018240	0.176810
SAS-L2	0.607038	0.001113	0.161399

5.3.2　実験 2 : SAS-L2 に必要なリソース使用量の調査

まず、認証情報のデータの長さを 32bit、ハッシュ関数として CRC-32 を使用した場合の CPU 使用率の計測結果を表 5.6 に示す。次に、認証情報のデータの長さを 64bit、ハッシュ関数として CRC-64 を使用した場合の CPU 使用率の計測結果を表 5.7 に示す。最後に、認証情報のデータの長さを 256bit、ハッシュ関数 (一方向性関数) として SHA-256 を使用した場合の CPU 使用率の計測結果を表 5.8 に示す。

表 5.6　各認証方式における認証部の CPU 使用率 (32bit, 単位%)

	CPU 使用率	**ユーザ CPU 使用率**	**システム CPU 使用率**
SAS-2	3.10	0.28	2.82
SAS-L2	3.05	0.16	2.88

表 5.7　各認証方式における認証部の CPU 使用率 (64bit, 単位%)

	CPU 使用率	**ユーザ CPU 使用率**	**システム CPU 使用率**
SAS-2	3.11	0.31	2.80
SAS-L2	2.90	0.18	2.72

表 5.8　各認証方式における認証部の CPU 使用率 (256bit, 単位%)

	CPU 使用率	**ユーザ CPU 使用率**	**システム CPU 使用率**
SAS-2	3.00	0.31	2.69
SAS-L2	2.85	0.15	2.70

5.4　評価と考察

全体の CPU 時間は、認証情報のサイズにかかわらず、SAS-2 よりも SAS-L2 の方が 0.1〜0.2 秒程度短いという結果が得られた (表 5.2)。また、SAS-2 における演算機能のみの CPU 時間を 100%とした場合、SAS-L2 では、32bit で約 0.2%、64bit で約 0.1%、256bit で約 6.1% の CPU 時間であり、大幅に計算量が削減されることを示す結果となった (表 5.3、5.4、5.5)。しかしながら、SAS-2 と SAS-L2 で計算量の理論値比較を行った場合、巡回冗長検査より演算負荷の大きい SHA-256 を採用する場合の方が計算量の差は大きくなるため、256bit の場合の計測結果は理論値通りの数値が得られなかった。原因としては、実装したプログラム内の演算回数が理論上の回数よりも多いことが挙げられる。C 言語で 64bit 以上の演算を実現するためには、数値を文字列に変換し、一文字ずつ文字コード同士の演算を行う方法や、認証情報を数値として演算できるデータ長のブロックに分割し、各ブロックごとに演算を行う

方法などが求められる。すなわち、64bit 以上の認証情報を数値として演算するために、理論上、1 回の演算で済む処理を複数回に分けて演算する必要がある。本研究における計測結果と理論値との差はこのような実装上における問題が影響している。

また、SAS-L2 の通信処理にかかる CPU 時間は、SAS-2 と比較して 1.6 倍強に増加している (表 5.3、5.4、5.5)。3 章で示したように、SAS-2、SAS-L2 の両アルゴリズムにおいて、通信回数は共通して 3 回だが、認証者側の通信の内訳が異なり、SAS-2 では送信回数が 2 回、受信回数が 1 回であることに対し、SAS-L2 では送信回数が 1 回、受信回数が 2 回である。本研究では送信に write 関数、受信に read 関数を使用した。現段階では未検証だが、実装したプログラム内で用いた送信用の関数、受信用の関数の処理負荷の違いが CPU 時間の差に影響している可能性がある。測定結果から受信用の関数の方が処理負荷が大きくなると仮定すると、SAS-2 よりも受信回数が多い SAS-L2 は、通信処理の負荷が大きくなっているであろう。

次に、CPU 使用率については、認証情報が 32bit の場合、SAS-L2 の方が 0.05%、認証情報が 64bit の場合は、SAS-L2 の方が 0.21%、認証情報が 256bit の場合は SAS-L2 の方が 0.15% 低い割合を示した (表 5.6、5.7、5.8)。アプリケーションによる CPU 使用率 (ユーザ CPU 使用率) とカーネルによる CPU 使用率 (システム CPU 使用率) に分解すると、認証情報のデータ長にかかわらず、アプリケーションによる CPU 使用率が低くなる傾向にあり、SAS-L2 における演算負荷削減の影響が測定結果に表れていた。一方で、カーネルによる CPU 使用率は認証情報が 32bit および 256bit の場合は SAS-L2 の方が高く、64bit の場合は SAS-L2 の方が低い割合を示しており、明確にその傾向が掴めなかった。カーネルによる CPU 使用率では通信処理や入出力処理が影響しており、通信処理の CPU 時間が SAS-L2 の方が長いことを踏まえると、入出力処理における負荷が SAS-L2 で削減されていなければ、カーネルによる CPU 使用率は SAS-2 よりも大きくなるのではないかと推測する。CPU への負荷については演算処理だけではなく、通信処理や入出力処理などの実現法について検討する必要がある。

最後に、メモリ使用量については、認証情報用変数の数および通信用に用意するバッファのサイズは SAS-2 と SAS-L2 で変化していないため、被認証者側で乱数生成や一方向性関数を適用する SAS-2 の方がメモリ使用量は大きくなる。本研究で実装したプログラムの場合、乱数生成用として/dev/urandom を使用しており、乱数を取得するためのバッファ(サイズは認証情報のデータと同じ) として確保できるメモリ領域が必要である。また、一方向性関数に SHA-256 を使用する場合は、パディング処理で追加する 256bit 分、初期ハッシュ値を格納するバッファ(32bit)8 個分、拡張メッセージ用のバッファ(32bit)64 個分、圧縮関数で使用するバッファ(32bit)8 個分を確保するメモリ領域も同様に必要となる。

第6章　結論と今後の課題

SAS-L2 は現段階では、組込みデバイスに対して、アルゴリズムを採用したプログラムを搭載した例が存在しない。そこで、本研究では、SAS-2 と SAS-L2 のアルゴリズムを採用したプログラムをそれぞれ、組込みデバイスの一つである Jetson Nano に搭載し、評価実験を行った。

実験結果から、SAS-L2 を用いることで認証時の演算処理の負荷が削減されていることは確認できたものの、カーネルが関わる処理、特に通信処理は不明点が多く、どの程度全体の CPU 使用率に影響するかは示せなかった。結論として、データ通信などの実現方法を検討する必要があるものの、SAS-L2 を適用させることによって、CPU やメモリなどのリソースの使用量を抑えながら、ワンタイムパスワード認証が行えるという点で、SAS-L2 を採用することの有意性は十分にある。

今後の課題について述べる。まずは、通信処理の負荷量に関する検討である。本研究ではソケットを介して送信に write 関数、受信に read 関数を使用した。しかし、そのほかの通信方法、関数、接続形態などについては未検討である。通信処理にかかる負荷が SAS-2 と同等もしくはそれ以下となる実装方法があれば、SAS-L2 の特徴である、演算処理削減の利点を生かすことが可能である。次に、SAS-2 から SAS-L2 に置き換えることによって新たに発生する脆弱性は存在しないかどうかを検討すること、また、運用時に通信経路の遮断などが原因で発生しうる認証情報の同期問題について、従来の SAS 認証方式については先行研究で成されており、同様に SAS-L2 における解決法を検討することといった、セキュリティ面に焦点を当てた課題も存在する。

謝辞

本研究を遂行するにあたり、常日頃より懇切丁寧な御指導を頂きました、甲斐博准教授に深く御礼申し上げます。本論文の作成に際し、詳細なるご検討、貴重な御教示を頂きました本学高橋寛教授ならびに王森レイ講師に深く御礼申し上げます。そして、本研究に際し、ご審査頂きました安藤和典准教授に深く御礼申し上げます。最後に、御支援いただいた本学情報工学科の諸先生方、研究室の皆様に厚く御礼申し上げます。

参考文献

［1］ 総務省，“5G が促すデジタル変革と新たな日常の構築，”
令和 2 年度版情報通信白書，pp.76-77，2020.

［2］ 清水明宏，“SAS-L ワンタイムパスワード認証方式について，” 高知工科大学,
preprint, 2020.

［3］ IPUSIRON，“暗号技術のすべて，” 翔泳社，2017.

［4］ Rafael Pass, Jed Liu, “Lecture 4: Hardness Amplification: From Weak to Strong OWFs,”
COM S 687 Introduction to Cryptography,
http://www.cs.cornell.edu/courses/cs687/2006fa/lectures/lecture4.pdf.

［5］ 金子敏信，“SHA-256/-384/-512 の評価報告，” CRYPTREC，pp.2-7，2006.

［6］ “巡回冗長検査 (CRC),” シニアエンジニアの庵,
https://sehermitage.web.fc2.com/crypto/CRC.html.

［7］ “CRC,” 通信用語の基礎知識, https://www.wdic.org/w/WDIC/CRC.

［8］ 裘カイ, 武田祐樹, 野上保之, 日下卓也, “CDM 系列と既存の擬似乱数生成器との比較,”
第 18 回情報科学技術フォーラム,
https://www.ieice.org/publications/conference-FIT-DVDs/FIT2019/data/pdf/L-011.pdf.

［9］ 太田愛里, “IoT に適したワンタイムパスワード認証方式に関する研究,” 卒業論文, 高
知工科大学, 2018.

［10］ Michael J. Donahoo, Kenneth L. Calvert, “TCP/IP ソケットプログラミング C 言語編,”
オーム社, 2003.

付 録A　参考：メモリ使用量の計測結果

参考として、認証部の処理に必要な物理メモリサイズの計測結果を表 A.1 に示す。

表 A.1　各認証方式の処理に必要な物理メモリのサイズ (単位 KB)

	32bit	**64bit**	**256bit**
SAS-2	516	524	1232
SAS-L2	488	520	516

付 録B　プログラムリスト

計算量を測定する際に用いたSAS-2認証及びSAS-L2認証のプログラムソースを以下に示す。

B.1　SAS-2認証

B.1.1　被認証者側の動作

```
#include <stdio.h>        /*printf()*/
#include <stdlib.h>       /*exit()*/
#include <string.h>       /*strlen()*/
#include <unistd.h>       /*close()*/
#include <sys/socket.h>   /*socket(),connect()*/
#include <sys/types.h>    /*setsockopt()*/
#include <arpa/inet.h>    /*struct sockaddr_in*/
#include <netinet/in.h>   /*inet_addr()*/
#include <openssl/sha.h>  /*SHA256*/

#define AUTHDATA_BYTE 32

int main(int argc, char **argv){

  //ソケット通信用
  register int sock;
  struct sockaddr_in server_addr;
  unsigned short servPort;
  char *servIP;

  //通信データ用バッファサイズ
  int rcvBufferSize=100*1024;
  int sendBufferSize=100*1024;

  //データ長
  unsigned short length;

  //各種フラグ
  unsigned short authent_flag=0;
  unsigned short overflow_flag=1;

  FILE *fp;

  //ループ用変数
  register int i;
  register int n;

  //認証情報
  unsigned char id[34]={'\0'};
  unsigned char pass[AUTHDATA_BYTE+2]={'\0'};
  unsigned char A[AUTHDATA_BYTE]={'\0'};
  unsigned char pre_A[AUTHDATA_BYTE]={'\0'};
  unsigned char A2[AUTHDATA_BYTE]={'\0'};
  unsigned char pre_A2[AUTHDATA_BYTE]={'\0'};
  unsigned char B[AUTHDATA_BYTE]={'\0'};
  unsigned char F[AUTHDATA_BYTE]={'\0'};
  unsigned char alpha[AUTHDATA_BYTE]={'\0'};
  unsigned char beta[AUTHDATA_BYTE]={'\0'};
  unsigned char gamma[AUTHDATA_BYTE]={'\0'};
  unsigned char Ni[AUTHDATA_BYTE]={'\0'};
  unsigned char Nii[AUTHDATA_BYTE]={'\0'};

  //暗号通信用
  unsigned char vernamData[AUTHDATA_BYTE+1]={'\0'};

  SHA256_CTX sha_ctx;

  if((argc<2) || (argc>3)){
    fprintf(stderr,"Usage: %s <Server IP> <Server Port>\n",argv[0]);
    exit(1);
  }

  servIP = argv[1]; //server IP 設定
  //ポート番号設定
  if(argc==4){
```

```
    servPort=atoi(argv[2]);    //ポート番号設定
  }
  else{
    servPort=7777;
  }

  //サーバ接続用ソケットを作成
  if ((sock = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("client: socket\n");
    exit(1);
  }

  //サーバアドレス用構造体の初期設定
  bzero((char *)&server_addr, sizeof(server_addr));
  server_addr.sin_family = PF_INET;
  server_addr.sin_addr.s_addr = inet_addr(servIP);
  server_addr.sin_port = htons(servPort);

  //サーバと接続
  if (connect(sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
    perror("client: connect\n");
    exit(1);
  }

  //バッファサイズを設定
  if(setsockopt(sock,SOL_SOCKET,SO_RCVBUF,&rcvBufferSize,sizeof(rcvBufferSize))<0){
    perror("setsockopt() failed\n");
    exit(1);
  }
  if(setsockopt(sock,SOL_SOCKET,SO_SNDBUF,&sendBufferSize,sizeof(sendBufferSize))<0){
    perror("setsockopt() failed\n");
    exit(1);
  }

  //初回登録フェーズ
  //フラグを受け取る
  read(sock, &authent_flag, sizeof(unsigned short));
  printf("flag=%d\n",authent_flag);

  if(authent_flag==0){
    printf("このユーザは未登録のため、初回登録を行います\n\n");

    //Ni を作成
    get_random(Ni,AUTHDATA_BYTE,AUTHDATA_BYTE);

    //Ni を保存
    if((fp=fopen("Ni_file.bin","wb"))==NULL){
      perror("Ni FILE OPEN ERROR\n");
      close(sock);
      exit(1);
    }
    fwrite(Ni,sizeof(Ni[0])*AUTHDATA_BYTE,1,fp);
    fclose(fp);

    printf("ID:");
    //最大 32 文字の入力を受け入れる
    if (fgets(id, 34, stdin)==NULL || id[0]=='\0' || id[0] == '\n'){
      perror("id:1-32char\n");
      close(sock);
      exit(1);
    }
    for(i=1; i<34; i++){
      if(id[i]=='\n'){
overflow_flag=0;
      }
    }
    if(overflow_flag==1){
      perror("id:1-32char\n");
      close(sock);
      exit(1);
    }
    overflow_flag=1;

    //改行を削除
    length=strlen(id);
    if(id[length-1]=='\n'){
      id[--length]='\0';
    }

    //送信する id の長さの情報を送信する
    write(sock, &length, sizeof(unsigned short));

    //ID を送信する
    write(sock, id, length);

    printf("PASS:");
    //最大 32 文字の入力を受け入れる
    if (fgets(pass, AUTHDATA_BYTE+2, stdin)==NULL || pass[0]=='\0' || pass[0]=='\n'){
      perror("PASS:1-32char\n");
      close(sock);
      exit(1);
    }
    for(i=1; i<AUTHDATA_BYTE+2; i++){
      if(pass[i]=='\n'){
overflow_flag=0;
      }
    }
```

```
  if(overflow_flag==1){
    perror("pass:1-32char\n");
    close(sock);
    exit(1);
  }
  overflow_flag=1;

  //改行を削除
  length=strlen(pass);
  if(pass[length-1]=='\n'){
    pass[--length]='\0';
  }

  //A を作成
  i=0;
  while(i<AUTHDATA_BYTE){
    pre_A[i]=pass[i]^Ni[i];
    i++;
  }

  SHA256_Init(&sha_ctx);
  SHA256_Update(&sha_ctx, pre_A, sizeof(pre_A));
  SHA256_Final(A, &sha_ctx);

  //A を送信する
  write(sock, A, AUTHDATA_BYTE);

  close(sock);
  exit(0);
}
else if(authent_flag==1){
  printf("このクライアントは登録済みです\n\n");
  authent_flag=0;
}
else{
 perror("flag error!\n");
 close(sock);
 exit(1);
}

printf("PASS:");
if (fgets(pass, AUTHDATA_BYTE+2, stdin)==NULL || pass[0]=='\0' || pass[0]=='\n'){
  perror("PASS:1-32char\n");
  close(sock);
  exit(1);
}
for(i=1; i<AUTHDATA_BYTE+2; i++){
  if(pass[i]=='\n'){
    overflow_flag=0;
  }
}
if(overflow_flag==1){
  perror("pass:1-32char\n");
  close(sock);
  exit(1);
}
overflow_flag=1;

//改行を削除
length=strlen(pass);
if(pass[length-1]=='\n'){
  pass[--length]='\0';
}

//認証フェーズ
if((fp=fopen("Ni_file.bin","rb"))==NULL){
  perror("Ni FILE OPEN ERROR\n");
  close(sock);
  exit(1);
}
fread(Ni,sizeof(Ni[0])*AUTHDATA_BYTE,1,fp);
fclose(fp);

//A の作成
i=0;
while(i<AUTHDATA_BYTE){
  pre_A[i]=pass[i]^Ni[i];
  i++;
}

SHA256_Init(&sha_ctx);
SHA256_Update(&sha_ctx, pre_A, sizeof(pre_A));
SHA256_Final(A, &sha_ctx);

//Ni+1 を作成
get_random(Nii,AUTHDATA_BYTE,AUTHDATA_BYTE);

//Ai+1 を作成
i=0;
while(i<AUTHDATA_BYTE){
  pre_A2[i]=pass[i]^Nii[i];
  i++;
}

SHA256_Init(&sha_ctx);
SHA256_Update(&sha_ctx, pre_A2, sizeof(pre_A2));
SHA256_Final(A2, &sha_ctx);
```

```
  SHA256_Init(&sha_ctx);
  SHA256_Update(&sha_ctx, A2, sizeof(A2));
  SHA256_Final(B, &sha_ctx);

  //alpha を作成
  i=0;
  while(i<AUTHDATA_BYTE){
    alpha[i]=A2[i]^(B[i]+A[i]);
    i++;
  }

  //beta を作成
  i=0;
  while(i<AUTHDATA_BYTE){
    beta[i]=B[i]^A[i];
    i++;
  }

  //alpha を送信する
  write(sock, alpha, AUTHDATA_BYTE);

  //beta を送信する
  write(sock, beta, AUTHDATA_BYTE);

  //gamma を受け取る
  read(sock, gamma, AUTHDATA_BYTE);

  //F=H(B) を作成する
  SHA256_Init(&sha_ctx);
  SHA256_Update(&sha_ctx, B, sizeof(B));
  SHA256_Final(F, &sha_ctx);

  for(i=0; i<AUTHDATA_BYTE; i++){
    if(gamma[i]!=F[i]){
      authent_flag=1;
    }
  }

  if(authent_flag==1){
    perror("gamma and F are not equal\n");
    perror("authentication error!\n");
    close(sock);
    exit(1);
  }
  else{
    printf("gamma and F equal\n");

    //次の認証情報をファイルに保存
    if((fp=fopen("Ni_file.bin","wb"))==NULL){
      perror("Ni FILE OPEN ERROR\n");
      close(sock);
      exit(1);
    }
    fwrite(Nii,sizeof(Nii[0])*AUTHDATA_BYTE,1,fp);
    fclose(fp);
  }

  //暗号通信フェーズ
  //バーナム暗号用メッセージを読み取り表示
  if((fp=fopen("Msg.txt","r"))==NULL){
    perror("Msg_FILE OPEN ERROR\n");
    close(sock);
    exit(1);
  }
  fgets(vernamData,AUTHDATA_BYTE+1,fp);
    for(i=1; i<AUTHDATA_BYTE+1; i++){
    if(vernamData[i]=='\0'){
      overflow_flag=0;
    }
  }
  if(overflow_flag==1){
    perror("vernamData:1-32char\n");
    close(sock);
    exit(1);
  }

  //暗号化
  i=0;
  length=strlen(vernamData);
  while(i<length){
    vernamData[i]^=A2[i];
    i++;
  }
  vernamData[i]='\0';

  //送信する msg の長さの情報を送信する
  write(sock, &length, sizeof(unsigned short));

  //msg を送信する
  write(sock, vernamData,length);
  close(sock);

  exit(0);
}
```

B.1.2 認証側の動作

```
#include <stdio.h>        /*printf()*/
#include <stdlib.h>       /*exit()*/
#include <string.h>       /*strcmp(),strcpy()*/
#include <unistd.h>       /*close()*/
#include <sys/types.h>    /*setsockopt()*/
#include <sys/socket.h>   /*socket(),bind(),connect(),accept()*/
#include <arpa/inet.h>    /*sockaddr_in,inet_ntoa()*/
#include <netinet/in.h>
#include <sys/time.h>     /*struct timeval*/
#include <openssl/sha.h>

#define MAXPENDING 5     //同時接続上限数
#define AUTHDATA_BYTE 32 //認証情報のデータ長

char str1[] = "Auth_file.bin";
FILE *fp;
SHA256_CTX sha_ctx;
/**
*登録済み User かどうかを調べる関数
*/
void UserDataSearch(int clntSock,struct sockaddr_in *clnt_addr){
  unsigned char dataLine[50]={'\0'}; //userdata の 1 行分を保持
  unsigned char search[20]={'\0'};   //探索する IP アドレスを保持
  char id[33]={'\0'};                //識別子を保持
  char *idp;                         //識別子用のポインタ
  char *ipp;                         //ip アドレス用のポインタ
  unsigned short authent_flag=0;

  if((fp=fopen("UserData.txt","r"))==NULL){
perror("User FILE OPEN ERROR\n");
exit(1);
  }
  else{
strcpy(search,inet_ntoa(clnt_addr->sin_addr));

while(fgets(dataLine,50,fp)){

  //探索対象を識別子に設定
  strtok(dataLine,",");
  ipp=strtok(NULL,"\n");
  if(strcmp(ipp, search)==0){
idp = strtok(dataLine,",");
authent_flag = 1;
break;
  }
  dataLine[0]='\0';
}
  }
  fclose(fp);

  //flag を送信
  write(clntSock, &authent_flag, sizeof(unsigned short));

  if(authent_flag == 0){
puts("識別子は見つかりませんでした");
InitRegistration(clntSock,search);
  }
  else if(authent_flag == 1){
puts("識別子が見つかりました");
strcpy(id,idp);

if(Authentication(clntSock,id)==0){
  VernamCipher(clntSock,id);
}
  }
  else{
perror("flag error\n");
exit(1);
  }
}

/**
*初期登録用関数
*/
void InitRegistration(int clntSock,unsigned char *clntIpAddr){
  char id[33]={'\0'};
  unsigned char addUserData[50]={'\0'};/*id,ip*/
  unsigned char A[AUTHDATA_BYTE]={'\0'};

  char filename1[50]={'\0'};

  register int i;
  unsigned short length;

  //ID のデータ長を記録
  read(clntSock, &length, sizeof(unsigned short));

  //ID を受け取る
  read(clntSock,id,length);

  if((fp=fopen("UserData.txt","a")) == NULL){
perror("User FILE OPEN ERROR\n");
exit(1);
  }
```

```
  sprintf(addUserData,"%s,%s\n",id,clntIpAddr);
  fprintf(fp,"%s",addUserData);
  fclose(fp);
  addUserData[0]='\0';

  //A を受け取る
  read(clntSock,A,AUTHDATA_BYTE);

  sprintf(filename1,"%s%s",id,str1);

  //A を保存する
  if((fp=fopen(filename1,"wb")) == NULL){
perror("Auth FILE OPEN ERROR\n");
exit(1);
  }
  fwrite(A,sizeof(A[0]),AUTHDATA_BYTE,fp);
  fclose(fp);

}

/**
*認証用関数
*/
int Authentication(int clntSock,char *id){
  unsigned char A[AUTHDATA_BYTE]={'\0'};
  unsigned char C[AUTHDATA_BYTE]={'\0'};
  unsigned char D[AUTHDATA_BYTE]={'\0'};
  unsigned char E[AUTHDATA_BYTE]={'\0'};
  unsigned char alpha[AUTHDATA_BYTE]={'\0'};
  unsigned char beta[AUTHDATA_BYTE]={'\0'};
  unsigned char gamma[AUTHDATA_BYTE]={'\0'};

  char filename1[50]={'\0'};
  unsigned short authent_flag=0;

  register int i;

  sprintf(filename1,"%s%s",id,str1); //User_Auth_file

  //A を読み取る
  if((fp=fopen(filename1,"rb")) == NULL){
printf("Ai FILE OPEN ERROR\n");
return -1;
  }
  fread(A,sizeof(A[0]),AUTHDATA_BYTE,fp);
  fclose(fp);

  //alpha を受け取る
  read(clntSock,alpha,AUTHDATA_BYTE);

  //beta を受け取る
  read(clntSock,beta,AUTHDATA_BYTE);

  //C を作成する
  i=0;
  while(i<AUTHDATA_BYTE){
C[i]=beta[i]^A[i];
i++;
  }

  //D を作成する
  i=0;
  while(i<AUTHDATA_BYTE){
D[i]=alpha[i]^(C[i]+A[i]);
i++;
  }

  //E を作成する
  SHA256_Init(&sha_ctx);
  SHA256_Update(&sha_ctx, D, sizeof(D));
  SHA256_Final(E, &sha_ctx);

  //C と E を比較
  for(i=0; i<AUTHDATA_BYTE; i++){
if(E[i]!=C[i]){
  authent_flag=1;
}
  }

  if(authent_flag==1){
puts("C and E are not equal");
perror("authentication error!\n");
return -1;
  }
  else if(authent_flag==0){
puts("C and E are equal");

//D(Ai+1 を保存する)
if((fp=fopen(filename1,"wb")) == NULL){
  perror("Auth FILE OPEN ERROR\n");
  return -1;
}
fwrite(D,sizeof(D[0]),AUTHDATA_BYTE,fp);
fclose(fp);

//gamma を作成
SHA256_Init(&sha_ctx);
```

```
SHA256_Update(&sha_ctx, C, sizeof(C));
SHA256_Final(gamma, &sha_ctx);

//gamma を送信
write(clntSock,gamma,AUTHDATA_BYTE);
  }
  return 0;
}

/**
*暗号通信用関数
*/
int VernamCipher(int clntSock,char *id){
  char filename1[50]={'\0'};
  unsigned char vernamKey[AUTHDATA_BYTE]={'\0'};
  unsigned char vernamData[AUTHDATA_BYTE+1]={'\0'};
  register int i;
  unsigned short length;

  sprintf(filename1,"%s%s",id,str1);

  //認証情報読み取り
  if((fp=fopen(filename1,"rb")) == NULL){
printf("Auth FILE OPEN ERROR\n");
exit(1);
  }
  fread(vernamKey,sizeof(vernamKey[0]),AUTHDATA_BYTE,fp);
  fclose(fp);

  //受け取る暗号文のデータ長を記録
  read(clntSock, &length, sizeof(unsigned short));

  //暗号文を受け取る
  read(clntSock, vernamData, length);

  //復号
  i=0;
  while(i<length){
vernamData[i]^=vernamKey[i];
i++;
  }
  vernamData[i]='\0';

  if((fp=fopen("Msg2.txt","w")) == NULL){
printf("Msg FILE OPEN ERROR\n");
exit(1);
  }
  fwrite(vernamData,sizeof(vernamData[0]),strlen(vernamData),fp);
  fclose(fp);

  return 0;
}

int main(int argc, char **argv){
  int *servSock;
  int maxDescriptor;
  fd_set watchFds;
  long timeout;
  struct timeval selTimeout;
  int running=1;
  int noPorts;
  register int port;
  unsigned short portNo;

  if(argc<3){
    fprintf(stderr,"Usage: %s <Timeout (secs.)> <Port 1>...\n",argv[0]);
    exit(1);
  }

  timeout=atol(argv[1]);
  noPorts=argc-2;

  servSock=(int*)malloc(noPorts*sizeof(int));

  maxDescriptor=-1;

  for(port=0; port<noPorts; port++){
    portNo=atoi(argv[port+2]);
    servSock[port]=CreateTCPServerSocket(portNo);

    if(servSock[port]>maxDescriptor){
      maxDescriptor=servSock[port];
    }
  }

  printf("Starting server:Hit return to shutdown\n");

  while(running){
    FD_ZERO(&watchFds);
    FD_SET(STDIN_FILENO,&watchFds);
    for(port=0; port<noPorts; port++){
      FD_SET(servSock[port],&watchFds);
    }

    selTimeout.tv_sec=timeout;
    selTimeout.tv_usec=0;
```

```
    //監視対象の FD のいずれかが読み込み可能になるまで待機
    if(select(maxDescriptor+1, &watchFds, NULL, NULL, &selTimeout)==0){
      fprintf(stderr,"No authent requests for %ld secs...Server still alive\n",timeout);
      break;
    }
    else{
      if(FD_ISSET(STDIN_FILENO,&watchFds)){
printf("Shutting down server\n");
getchar();
running=0;
      }
      for(port=0; port<noPorts; port++){
if(FD_ISSET(servSock[port],&watchFds)){
  printf("Request on port %d: ",atoi(argv[port+2]));
  AcceptTCPConnection(servSock[port]);
}
      }
    }
  }

  for(port=0; port<noPorts; port++){
    printf("client Disconnected and closed port %d\n",atoi(argv[port+2]));
    close(servSock[port]);
  }
  free(servSock);
  return 0;
}
```

B.2 SAS-L2 認証

B.2.1 被認証者側の動作

```
#include <stdio.h>       /*printf()*/
#include <stdlib.h>      /*exit()*/
#include <string.h>      /*strlen(),strcmp()*/
#include <unistd.h>      /*close()*/
#include <sys/types.h>   /*setsockopt()*/
#include <arpa/inet.h>    /*struct sockaddr_in*/
#include <sys/socket.h>  /*socket(),connect(),read(),write()*/
#include <netinet/in.h>  /*inet_addr()*/

//認証情報のデータ長
#define AUTHDATA_BYTE 32

int main(int argc, char **argv){

  //ソケット通信用
  register int sock;
  struct sockaddr_in server_addr;
  unsigned short servPort;
  char *servIP;

  //通信データ用バッファサイズ
  int rcvBufferSize=100*1024;
  int sendBufferSize=100*1024;

  //データ長
  unsigned short length;

  //認証用フラグ
  unsigned short authent_flag=0;

  FILE *fp;

  //ループ用変数
  register int i;
  register int n;

  //認証情報
  unsigned char B[AUTHDATA_BYTE]={'\0'};
  unsigned char C[AUTHDATA_BYTE]={'\0'};
  unsigned char D[AUTHDATA_BYTE]={'\0'};
  unsigned char alpha[AUTHDATA_BYTE]={'\0'};
  unsigned char gamma[AUTHDATA_BYTE]={'\0'};
  unsigned char authData[2][AUTHDATA_BYTE]={'\0'};

  //暗号通信用
  unsigned char vernamData[AUTHDATA_BYTE+1]={'\0'};

  //実行時の入力エラー
  if((argc<2) || (argc>3)){
    fprintf(stderr,"Usage: %s <Server IP> <Server Port>\n",argv[0]);
    exit(1);
  }

  servIP = argv[1]; //server IP 設定
  //ポート番号設定
  if(argc==3){
```

```
  servPort=atoi(argv[2]);
}
else{
  servPort=7777;
}

//サーバ接続用ソケットを作成
if ((sock = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
  perror("client: socket\n");
  exit(1);
}

//サーバアドレス用構造体の初期設定
bzero((char *)&server_addr, sizeof(server_addr));
server_addr.sin_family = PF_INET;
server_addr.sin_addr.s_addr = inet_addr(servIP);
server_addr.sin_port = htons(servPort);

//サーバと接続
if (connect(sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
  perror("client: connect\n");
  exit(1);
}

//バッファサイズを設定
if(setsockopt(sock,SOL_SOCKET,SO_RCVBUF,&rcvBufferSize,sizeof(rcvBufferSize))<0){
  perror("setsockopt() failed\n");
  exit(1);
}
if(setsockopt(sock,SOL_SOCKET,SO_SNDBUF,&sendBufferSize,sizeof(sendBufferSize))<0){
  perror("setsockopt() failed\n");
  exit(1);
}

//初回登録フェーズ

//フラグを受け取る
read(sock, &authent_flag, sizeof(unsigned short));

if(authent_flag==0){
  printf("このユーザは未登録のため、初回登録を行います\n\n");

  //A,M を受け取る
  read(sock, &authData[0], AUTHDATA_BYTE);
  read(sock, &authData[1], AUTHDATA_BYTE);

  if((fp=fopen("Auth_file.bin","wb"))==NULL){
    perror("Auth FILE OPEN ERROR\n");
    close(sock);
    exit(1);
  }
  fwrite(authData,sizeof(authData[0][0])*AUTHDATA_BYTE,2,fp);
  fclose(fp);

  close(sock);
  exit(0);
}

else if(authent_flag==1){
  printf("このクライアントは登録済みです\n\n");
  authent_flag=0;
}

else{
  perror("flag error!\n");
  close(sock);
  exit(1);
}

//認証フェーズ
//alpha を受け取る
read(sock, alpha, AUTHDATA_BYTE);

//ファイルから A と M を読み取る
if((fp=fopen("Auth_file.bin","rb"))==NULL){
  perror("Auth FILE OPEN ERROR\n");
  close(sock);
  exit(1);
}
fread(authData,sizeof(authData[0][0])*AUTHDATA_BYTE,2,fp);
fclose(fp);

//B を作成する
i=0;
while(i<AUTHDATA_BYTE){
  B[i]=alpha[i]^authData[0][i]^authData[1][i];
  i++;
}

//C を作成する
i=0;
while(i<AUTHDATA_BYTE){
  C[i]=authData[0][i]+B[i];
  i++;
}

//C を送信する
```

```
  write(sock, C, AUTHDATA_BYTE);

  //gamma を受け取る
  read(sock, gamma, AUTHDATA_BYTE);

  //Mi+1 を作成する
  i=0;
  while(i<AUTHDATA_BYTE){
    authData[1][i]=authData[0][i]+authData[1][i];
    i++;
  }

  //D を作成する
  i=0;
  while(i<AUTHDATA_BYTE){
    D[i]=authData[0][i]^authData[1][i];
    i++;
  }

  for(i=0; i<AUTHDATA_BYTE; i++){
    if(gamma[i]!=D[i]){
      authent_flag=1;
    }
  }

  if(authent_flag==1){
    perror("gamma and D are not equal\n");
    perror("authentication error!\n");
    close(sock);
    exit(1);
  }
  else{
    printf("gammma and D equal\n");
    memcpy(authData,B,sizeof(B));

    //次の認証情報をファイルに保存
    if((fp=fopen("Auth_file.bin","wb"))==NULL){
      perror("Auth FILE OPEN ERROR\n");
      close(sock);
      exit(1);
    }
    fwrite(authData,sizeof(authData[0][0])*AUTHDATA_BYTE,2,fp);
    fclose(fp);
  }

  //暗号通信フェーズ
  //バーナム暗号用メッセージを読み取り表示
  if((fp=fopen("Msg.txt","r"))==NULL){
    perror("Msg_FILE OPEN ERROR\n");
    close(sock);
    exit(1);
  }
  fgets(vernamData,AUTHDATA_BYTE+1,fp);

  //暗号化
  i=0;
  length=strlen(vernamData);
  while(i<length){
    vernamData[i]^=B[i];
    i++;
  }
  vernamData[i]='\0';

  //送信する msg の長さの情報を送信する
  write(sock, &length, sizeof(unsigned short));

  //msg を送信する
  write(sock, vernamData,length);
  close(sock);

  exit(0);
}
```

B.2.2 認証側の動作

```
#include <stdio.h>      /*printf()*/
#include <stdlib.h>     /*exit()*/
#include <string.h>     /*strcmp(),strcpy()*/
#include <unistd.h>     /*close()*/
#include <sys/types.h>  /*setsockopt()*/
#include <sys/socket.h> /*socket(),bind(),connect(),accept()*/
#include <arpa/inet.h>  /*sockaddr_in,inet_ntoa()*/
#include <netinet/in.h>
#include <sys/time.h>   /*struct timeval*/
#include <openssl/sha.h>

#define MAXPENDING 5              //同時接続上限数
#define AUTHDATA_BYTE 32          //認証情報のデータ長

char str1[] = "Auth_file.bin";
FILE *fp;
SHA256_CTX sha_ctx;
```

```
/**
*登録済み User かどうか調べる関数
*/
void UserDataSearch(int clntSock,struct sockaddr_in *clnt_addr){
  unsigned char dataLine[50]={'\0'}; //userdata の 1 行分を保持
  unsigned char search[20]={'\0'};   //探索する IP アドレスを保持
  char id[AUTHDATA_BYTE+1]={'\0'};   //識別子を保持
  char *idp;                         //識別子用のポインタ
  char *ipp;                         //ip アドレス用のポインタ
  unsigned short authent_flag=0;     //処理分岐用フラグ

  if((fp=fopen("UserData.txt","r"))==NULL){
    perror("User FILE OPEN ERROR\n");
    exit(1);
  }
  else{
    strcpy(search,inet_ntoa(clnt_addr->sin_addr));

    while(fgets(dataLine,50,fp)){

      //探索対象を識別子に設定
      strtok(dataLine,",");
      ipp=strtok(NULL,"\n");

      if(strcmp(ipp, search)==0){
idp = strtok(dataLine,",");
authent_flag = 1;
break;
      }
      dataLine[0]='\0';
    }
  }
  fclose(fp);

  //flag を送信
  write(clntSock,&authent_flag, sizeof(unsigned short));

  if(authent_flag==0){
    puts("識別子は見つかりませんでした");
    InitRegistration(clntSock,search);
  }
  else if(authent_flag == 1){
    printf("識別子が見つかりました\n");
    strcpy(id,idp);

    if(Authentication(clntSock,id)==0){
      VernamCipher(clntSock,id);
    }
  }
  else{
    perror("flag error\n");
    exit(1);
  }
}

/**
*初期登録用関数
*/
void InitRegistration(int clntSock,unsigned char *clntIpAddr){

  //認証情報
  char id[AUTHDATA_BYTE+2]={'\0'};
  unsigned char pre_A[AUTHDATA_BYTE]={'\0'};
  unsigned char Ni[AUTHDATA_BYTE]={'\0'};
  unsigned char authData[2][AUTHDATA_BYTE]={'\0'};

  unsigned char addUserData[50]={'\0'};/*(id),(ip address)*/

  //オーバーヘッド検出用フラグ
  unsigned short overflow_flag=1;

  char filename1[50]={'\0'};

  //ループ用変数
  register int i;

  //データ長
  unsigned short length;

  puts("識別子を設定してください");
  //標準入力を受け取る
  if(fgets(id,AUTHDATA_BYTE+2,stdin)==NULL || id[0]=='\0' || id[0]=='\n'){
    perror("識別子:1-32char\n");
    exit(1);
  }
  for(i=1; i<AUTHDATA_BYTE+2; i++){
    if(id[i]=='\n'){
      overflow_flag=0;
    }
  }
  if(overflow_flag==1){
    perror("識別子:1-32char\n");
    exit(1);
  }
  overflow_flag=1;
```

```
  //改行を削除
  length=strlen(id);
  if(id[length-1]=='\n'){
    id[--length]='\0';
  }

  if((fp=fopen("UserData.txt","a")) == NULL){
    perror("FILE OPEN ERROR\n");
    exit(1);
  }

  sprintf(addUserData,"%s,%s\n",id,clntIpAddr);
  fprintf(fp,"%s",addUserData);
  fclose(fp);
  addUserData[0]='\0';

  //Ni を作成
  get_random(Ni,AUTHDATA_BYTE,AUTHDATA_BYTE);

  //Mi 作成
  get_random(authData[1],AUTHDATA_BYTE,AUTHDATA_BYTE);

  //A を作成
  i=0;
  while(i<AUTHDATA_BYTE){
    pre_A[i]=id[i]^Ni[i];
    i++;
  }

  SHA256_Init(&sha_ctx);
  SHA256_Update(&sha_ctx, pre_A, sizeof(pre_A));
  SHA256_Final(authData[0], &sha_ctx);

  //A,M を送信
  write(clntSock,&authData[0],AUTHDATA_BYTE);
  write(clntSock,&authData[1],AUTHDATA_BYTE);

  sprintf(filename1,"%s%s",id,str1); //User_Auth_file

  //A,M を保存
  if((fp=fopen(filename1,"wb")) == NULL){
    perror("Auth FILE OPEN ERROR\n");
    exit(1);
  }
  fwrite(authData,sizeof(authData[0][0])*AUTHDATA_BYTE,2,fp);
  fclose(fp);
}

/**
*認証用関数
*/
int Authentication(int clntSock,char *id){

  //認証情報
  unsigned char pre_A2[AUTHDATA_BYTE]={'\0'};
  unsigned char A2[AUTHDATA_BYTE]={'\0'};
  unsigned char C[AUTHDATA_BYTE]={'\0'};
  unsigned char Nii[AUTHDATA_BYTE]={'\0'};
  unsigned char alpha[AUTHDATA_BYTE]={'\0'};
  unsigned char beta[AUTHDATA_BYTE]={'\0'};
  unsigned char gamma[AUTHDATA_BYTE]={'\0'};
  unsigned char authData[2][AUTHDATA_BYTE]={'\0'}; //[0]:Ai,[1]:Mi

  //ユーザごとに用意された認証情報ファイルの名前を格納する配列
  char filename[50]={'\0'};

  //認証用フラグ
  unsigned short authent_flag=0;

  //ループ用変数
  register int i;

  sprintf(filename,"%s%s",id,str1); //Username+"Auth_file.bin"

  //Ni+1 の作成
  get_random(Nii,AUTHDATA_BYTE,AUTHDATA_BYTE);

  //Ai+1 の作成
  i=0;
  while(i<AUTHDATA_BYTE){
    pre_A2[i]=id[i]^Nii[i];
    i++;
  }
  SHA256_Init(&sha_ctx);
  SHA256_Update(&sha_ctx, pre_A2, sizeof(pre_A2));
  SHA256_Final(A2, &sha_ctx);

  //ファイルから A と M を読み取る
  if((fp=fopen(filename,"rb")) == NULL){
    printf("Auth FILE OPEN ERROR\n");
    return -1;
  }
  fread(authData,sizeof(authData[0][0])*AUTHDATA_BYTE,2,fp);
  fclose(fp);

  //alpha を作成する
  i=0;
```

```
  while(i<AUTHDATA_BYTE){
    alpha[i]=authData[0][i]^A2[i]^authData[1][i];
    i++;
  }

  //beta を作成する
  i=0;
  while(i<AUTHDATA_BYTE){
    beta[i]=authData[0][i]+A2[i];
    i++;
  }

  //Mi+1 を作成する
  i=0;
  while(i<AUTHDATA_BYTE){
    authData[1][i]=authData[0][i]+authData[1][i];
    i++;
  }

  //alpha を送信する
  write(clntSock, alpha, AUTHDATA_BYTE);

  //C を受け取る
  read(clntSock,C,AUTHDATA_BYTE);

  for(i=0; i<AUTHDATA_BYTE; i++){
    if(beta[i]!=C[i]){
      authent_flag=1;
    }
  }

  if(authent_flag==1){
    puts("beta and C are not equal");
    perror("authentication error!\n");
    return -1;
  }
  else{
    puts("beta and C are equal");

    //gamma を作成する
    i=0;
    while(i<AUTHDATA_BYTE){
      gamma[i]=authData[0][i]^authData[1][i];
      i++;
    }

    //A2(Ai+1) と Mi+1 を保存する
    memcpy(authData,A2,sizeof(A2));
    if((fp=fopen(filename,"wb")) == NULL){
      perror("Auth FILE OPEN ERROR\n");
      return -1;
    }
    fwrite(authData,sizeof(authData[0][0])*AUTHDATA_BYTE,2,fp);
    fclose(fp);

    //gamma を送信する
    write(clntSock, gamma, AUTHDATA_BYTE);
  }
  return 0;
}

/**
*暗号通信用関数
*/
int VernamCipher(int clntSock,char *id){

  //バーナム暗号情報
  unsigned char vernamKey[AUTHDATA_BYTE]={'\0'}; //秘密鍵
  unsigned char vernamData[AUTHDATA_BYTE+1]={'\0'};

  //ユーザごとに用意された認証情報ファイルの名前を格納する配列
  char filename[50]={'\0'};

  //ループ用変数
  register int i;

  //暗号文のデータ長
  unsigned short length;

  sprintf(filename,"%s%s",id,str1);

  //認証情報読み取り
  if((fp=fopen(filename,"rb")) == NULL){
    printf("Auth FILE OPEN ERROR\n");
    exit(1);
  }
  fread(vernamKey,sizeof(vernamKey[0])*AUTHDATA_BYTE,1,fp);
  fclose(fp);

  //受け取る暗号文のデータ長を取得
  read(clntSock, &length, sizeof(unsigned short));

  //暗号文を受け取る
  read(clntSock, vernamData, length);

  //復号
  i=0;
```

```
  while(i<length){
    vernamData[i]^=vernamKey[i];
    i++;
  }
  vernamData[i]='\0';

  //復号結果をファイルに保存
  if((fp=fopen("Msg2.txt","w")) == NULL){
    printf("Auth FILE OPEN ERROR\n");
    exit(1);
  }
  fwrite(vernamData,sizeof(vernamData[0]),strlen(vernamData),fp);
  fclose(fp);

  return 0;
}

int main(int argc, char **argv){
  int *servSock;
  int maxDescriptor;
  fd_set watchFds;
  long timeout;
  struct timeval selTimeout;
  int running=1;
  int noPorts;
  register int port;
  unsigned short portNo;

  if(argc<3){
    fprintf(stderr,"Usage: %s <Timeout (secs.)> <Port 1>...\n",argv[0]);
    exit(1);
  }

  timeout=atol(argv[1]);
  noPorts=argc-2;

  servSock=(int*)malloc(noPorts*sizeof(int));

  maxDescriptor=-1;

  for(port=0; port<noPorts; port++){
    portNo=atoi(argv[port+2]);
    servSock[port]=CreateTCPServerSocket(portNo);

    if(servSock[port]>maxDescriptor){
      maxDescriptor=servSock[port];
    }
  }

  printf("Starting server:Hit return to shutdown\n");

  while(running){
    FD_ZERO(&watchFds);
    FD_SET(STDIN_FILENO,&watchFds);
    for(port=0; port<noPorts; port++){
      FD_SET(servSock[port],&watchFds);
    }

    selTimeout.tv_sec=timeout;
    selTimeout.tv_usec=0;

    //監視対象の FD のいずれかが読み込み可能になるまで待機
    if(select(maxDescriptor+1, &watchFds, NULL, NULL, &selTimeout)==0){
      fprintf(stderr,"No authent requests for %ld secs...Server still alive\n",timeout);
      break;
    }
    else{
      if(FD_ISSET(STDIN_FILENO,&watchFds)){
      printf("Shutting down server\n");
      getchar();
      running=0;
      }
      for(port=0; port<noPorts; port++){
      if(FD_ISSET(servSock[port],&watchFds)){
        printf("Request on port %d: ",atoi(argv[port+2]));
        AcceptTCPConnection(servSock[port]);
      }
      }
    }
  }

  for(port=0; port<noPorts; port++){
    printf("client Disconnected and closed port %d\n",atoi(argv[port+2]));
    close(servSock[port]);
  }
  free(servSock);
  return 0;
}
```

B.3 その他のプログラム

```
/**
*Linux カーネルの乱数生成器から乱数を取得するプログラム
*/
#define DEV_RANDOM "/dev/urandom"
void get_random(char *const buf, const int bufLen, const int len){
  //初期エラー
  if(len > bufLen){
    perror("buffer size is small\n");
    exit(1);
  }

  int fd = open(DEV_RANDOM, O_RDONLY);
  if(fd == -1){
    printf("can not open %s\n", DEV_RANDOM);
    exit(1);
  }

  int r = read(fd,buf,len);
  if(r < 0){
    perror("can not read\n");
    exit(1);
  }
  if(r != len){
    perror("can not read\n");
    exit(1);
  }

  close(fd);
}

/**
*CRC-32 を演算する関数
*/
unsigned long crc32(const unsigned char *s,int len){
  unsigned int crc_init = 0;
  unsigned int crc =0;

  crc = crc_init ^ 0xFFFFFFFF;
  for(;len--;s++){
    crc = ((crc >> 8)&0x00FFFFFF) ^ crc32tab[(crc ^ (*s)) & 0xFF];
  }
  return crc ^ 0xFFFFFFFF;
}

/**
*CRC-64 を演算する関数
*/
unsigned long crc64(const unsigned char *s, unsigned long len) {
  unsigned long crc=0;
  unsigned long j;

  for (j = 0; j < len; j++) {
    unsigned int byte = s[j];
    crc = crc64_tab[(crc ^ byte)&0xFF] ^ (crc >> 8);
  }
  return crc;
}

void AcceptTCPConnection(int servSock){

  //ソケット通信用
  int clntSock;
  unsigned int clntLen;
  struct sockaddr_in clnt_addr;

  //通信データ用バッファサイズ
  int rcvBufferSize=100*1024;
  int sendBufferSize=100*1024;

  clntLen=sizeof(clnt_addr);

  if((clntSock=accept(servSock,(struct sockaddr *)&clnt_addr,&clntLen))<0){
    perror("accept() failed\n");
    exit(1);
  }

  printf("Accepted connection from a client %s\n",inet_ntoa(clnt_addr.sin_addr));

  //バッファサイズを設定
  if(setsockopt(clntSock,SOL_SOCKET,SO_RCVBUF,&rcvBufferSize,sizeof(rcvBufferSize))<0){
    perror("setsockopt() failed\n");
    exit(1);
  }
  if(setsockopt(clntSock,SOL_SOCKET,SO_SNDBUF,&sendBufferSize,sizeof(sendBufferSize))<0){
    perror("setsockopt() failed\n");
    exit(1);
  }

  UserDataSearch(clntSock,&clnt_addr);
  close(clntSock);
}

int CreateTCPServerSocket(unsigned short port){
  int sock;
```

```
  struct sockaddr_in server_addr;

  if((sock=socket(PF_INET,SOCK_STREAM,IPPROTO_TCP))<0){
    perror("socket() failed\n");
    exit(1);
  }

  //構造体 server_addr の設定
  bzero((char *)&server_addr, sizeof(server_addr));
  server_addr.sin_family = PF_INET;
  server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  server_addr.sin_port = htons(port);

  //サーバの IP アドレスとポート番号をバインド
  if (bind(sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
    perror("bind() failed\n");
    exit(1);
  }

  //接続待機開始
  if(listen(sock, MAXPENDING)<0){
    perror("listen() failed\n");
    exit(1);
  }

  return sock;
}
```