ストリーミング処理によるWebサービスおよびWebサービスセキュリティの軽量実装

寺口正義[†] 山口裕美[†]
西海聡子[†] 伊藤貴之[†]

本論文では，ユビキタス時代の到来に備え，モバイル機器，オフィス機器のような計算資源の限られた情報機器上でも Web サービスおよび Web サービスセキュリティを実現するために，高速かつ省メモリのストリーミング処理による軽量実装について述べる．エミュレータを用いた実験により，著者らのプロトタイプが，DOM による中間構造を用いた実装と比較して，Web サービスの基本処理において 2.5〜4.0 倍高速にかつ 14%〜42%少ないメモリ使用量で動作することを確かめた．同様に Web サービスセキュリティの処理において，送信時 1.5 倍，受信時 1.8〜2.0 倍高速に動作することも分かった．

A Stream-based Implementation of Web Services and Web Services Security

Masayoshi Teraguchi,[†] Yumi Yamaguchi,[†] Akiko Nishikai[†] and Takayuki Itoh[†]

In this paper, we present a stream-based light-weight implementation of Web Services and Web Services Security for every IT devices with limited CPU resources and a limited amount of memory, such as mobile devices and office machines. Compared to an existing implementation that uses a DOM-like data structure for the intermediate processing, our prototype runs 2.5–4.0 times faster and requires 14%–42% less memory for encoding and decoding of SOAP messages. For Web Services Security processing, it also runs 1.5 times faster when processing a request SOAP message and runs 1.8–2.0 times faster when processing a response SOAP message.

1. はじめに

Web サービスとは，自己記述型の検索可能な汎用サービスを分散ソフトウェア部品として扱い，プラットフォームやアクセスするデバイスの種類を問わず，組織の枠を越えてこれらのサービスを柔軟に連携させる仕組みであり，XML 技術[1] を利用した通信プロトコル SOAP[2] 等の標準技術からなる．Web サービスセキュリティ（WS-Security[3]）とは，Web サービスにて通信される SOAP メッセージの完全性（通信途中で改竄されていないこと）や秘匿性（通信途中で覗き見されないこと）をメッセージレベルで End-to-End に保証するために，SOAP メッセージに署名や暗号化を適用するための仕様である．

これらの技術的流れをうけ，各社から Web サービスおよび WS-Security に対応したサーバや PC 向けの製品が公開されているが，文献 4) でも指摘されているように，WS-Security による性能低下が問題となっており，性能改善は 1 つの重要な課題である．

一方で，モバイル機器，オフィス機器等の各種情報機器の性能向上にともない，ユビキタスコンピューティング[5] の世界が現実味を帯びてきている．ユビキタスコンピューティングとは「利用者が意識することなく，いつでもどこでもネットワークに結合されたデバイスを通じて，様々な情報にアクセスできる」環境を指しており，計算資源の限られた各種情報機器でも自律的かつ安全に情報交換するための技術が必要になる．Web サービスおよび WS-Security はその目的に対して最適な技術である．

そこで，著者らは計算資源の限られた各種情報機器上でも Web サービスおよび WS-Security を高速かつ省メモリで実現するために，文献 6) の手法を応用した SAX[7] を用いたストリーミング処理に基づく軽量

† 日本アイ・ビー・エム株式会社東京基礎研究所
Tokyo Research Laboratory, IBM Japan, Ltd.

実装を試みた．各種情報機器の動作環境のほとんどがJ2ME[8)]☆であることを考慮し，著者らは J2ME の最小構成にあたる CLDC[9)] で動作するように実装した．文献 10) にもあるように，J2ME 上で動作する XML パーザは市場に出ていないため，独自の XML パーザを利用し，同様に WS-Seuciry の処理を行うためのセキュリティの基本ライブラリも独自のものを利用した．さらに，WSTKMD[11)] に搭載されている DOM[12)] を用いる実装と著者らの実装との性能評価実験を行うことで，著者らの実装が高速，省メモリで動作することを確かめる．また，市販の CLDC に対応した携帯電話上で性能評価実験を行うことで，実際の CLDC 環境上で動作することを確認するとともに，その性能を測る．

本論文の構成は次のとおりである．2 章で WS-Security の現状と関連研究について述べ，3 章で著者らが提案するストリーミング処理に基づく実装方法について述べる．4 章では，著者らが開発したプロトタイプの性能評価実験，市販の携帯電話実機上での性能評価実験について述べ，5 章で本論文のまとめを述べる．

2. WS-Security の現状

2.1 WS-Security の仕様とサブセット

WS-Security[3)] は，SOAP メッセージの署名および暗号化，それに SOAP メッセージにセキュリティトークンを付加する方法を規定する．署名には XML 電子署名[13)]，暗号化には XML 暗号化[14)] を利用する．**図 1** に署名，暗号化された SOAP メッセージの一例を示す．図 1 では，左側に署名，暗号化される前の SOAP メッセージを，右側に署名，暗号化された後の SOAP メッセージを示している．署名処理では，署名対象 S1 に署名し，受信者が署名を検証するために必要な情報 S2 を SOAP ヘッダに埋め込む．署名検証に必要な情報としては，セキュリティトークンに関する情報（<BinarySecurityToken>要素），署名情報（<SignedInfo>要素），署名値（<SignatureValue>要素）や署名対象（<Reference>要素）等が含まれる．暗号化処理では，暗号化対象 E1 を暗号化し，<EncryptedData>要素 E1' で置き換える．また，受信者が復号化するために必要な情報 E2 を SOAP ヘッダに埋め込む．復号化に必要な情報としては，鍵に関する情報（<EncryptedKey>要素）や暗号化対象

☆ “Java” およびすべての Java 関連の商標およびロゴは Sun Microsystems, Inc. の米国およびその他の国における商標または登録商標．

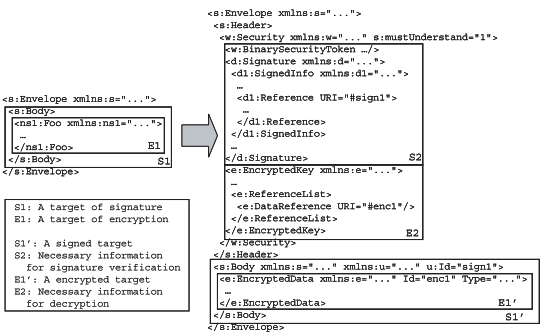


図 1 署名，暗号化された SOAP メッセージの一例
Fig. 1 A sample of the signed and encrypted message.

（<ReferenceList>要素）等が含まれる．セキュリティトークンは，たとえば送信者認証情報や署名検証に必要な公開鍵証明書の伝播に用いる．

現在，WS-Security の標準化と並行して，WS-Security のサブセットにあたる SOAP Message Security: Minimalist Profile（MProf[15)]）の標準化が進められている．MProf では，WS-Security のストリーミング処理を実現するためにいくつかの制約を設けている．以下に MProf で定められている制約のうち，今回の実装で仮定した制約事項のみを列挙する．

(1) 暗号化対象および署名対象は基本的に前方参照のみとする．
(2) セキュリティトークンは基本的に後方参照のみとする．
(3) XML の正規化は送信側のみで行う．ただし，XML 正規化として排他的正規化（Exclusive XML Canonicalization[16)]）のみを利用する．

2.2 J2ME 環境での WS-Security

各種情報機器の中には，携帯電話や PDA 等の小型機器も含まれている．これらの小型機器は，一般の計算機に比べて処理速度が大幅に遅く，また実行時メモリ搭載量も大幅に小さい．小型機器のほとんどがもつ J2ME 環境上で Web サービスおよび WS-Security を実装するためには，処理の高速化やメモリ使用量の低減が大きな要因となる．Imamura らは XML 暗号化をストリーミング処理することによる高速化や省メモリに取り組んでおり[6)]，XML 暗号化をストリーミング処理するうえで，暗号化が多重に適用されうるという難しさを XML パーザ内部で暗号化処理器を鎖状に結合させることにより解決している．ただし，Imamura らの手法は XML 署名を考慮しておらず，そのままでは WS-Security のストリーミング処理に反映できない．また，文献 10) でも指摘されているように，現状 J2ME 上で動作する標準仕様に準拠した XML パーザ

```
<s:Envelope xmlns:s="...">
 …
 <s:Body xmlns:s="..." xmlns:u1="http://AAA" u1:Id="sign1">
  …
  <ns1:Foo xmlns:ns1="..." xmlns:u2="http://AAA" u2:Id="sign2">
   …
  </ns1:Foo>                                              S2
  …
 </s:Body>                                                S1
</s:Envelope>
```

図 2 署名対象が入れ子になる一例
Fig. 2 A sample of nesting of signature targets.

が市場に出回っておらず，XML パーザ内部に処理系を持たせるのは困難である．一方で，XML 電子署名のストリーミング処理を提案している関連研究を見つけることはできなかった．

2.3 WS-Security のストリーミング処理の難しさ

XML 暗号化をストリーミング処理するうえでの難しさは暗号化の多重適用にあるが，Imamura らの手法と同様に処理系を再帰的に結合することで解決可能である．XML 電子署名をストリーミング処理するうえでの難しさは XML 正規化にある．2.1 節で示した MProf の制約 (3) を実現するためには，**図 2** のように，署名対象 S1，S2 が入れ子になった場合でも S1，S2 ともに正規化された状態を維持する必要がある．この場合，同一の名前空間 “http://AAA” の接頭辞に同じもの（たとえば，両方 $u1$ にする）を利用してはならず，必ず接頭辞を変えなければならない．一方，XML 暗号化と XML 電子署名を両方適用する場合には，暗号化，署名の処理順序が問題となる．図 1 の例では，メッセージ構築時に E1 の暗号化，S1 の署名の順に処理すれば問題ないが，メッセージ解析時には S1’ の署名検証，E1’ の復号化の順で処理する必要がある．したがって，S1’ の署名検証が終わるまで E1’ の復号化を待つような仕組みが必要となる．

そこで，著者らは WS-Security の実装において，上記で述べた処理の難しさをすべて解決可能なストリーミング処理による新たな実装方法を提案する．ただし，ストリーミング処理を実現するために SAX を用いることとし，2.1 節で述べた MProf の 3 つの制約すべてを仮定している．また，本論文では扱う SOAP メッセージ（XML）のフォーマットは UTF-8 [17] のみを仮定しているが，UTF-16 [18] への対応も，UTF-8 と UTF-16 間の変換が多く発生することになるが，同様に実現可能であると考える．

さらに，著者らは Web サービスの実装に関しても，JSR172 [19] に準拠したうえで，できる限りストリーミング処理を適用することにより，実装全体の処理を効率化することを目指す．ただし，JSR172 では SOAP Attachment は対象としていないため，WS-Security の実装でも SOAP Attachment を考慮しないこととした．

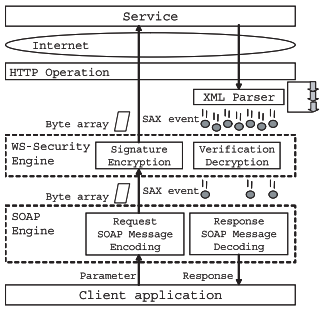


図 3 ストリーミング処理による軽量実装のアーキテクチャ
Fig. 3 An architecture of stream-based implementation.

3. ストリーミング処理による Web サービスおよび WS-Security の実装

3.1 軽量実装のアーキテクチャ

著者らが想定する SAX を用いたストリーミング処理による Web サービスおよび WS-Security の軽量実装のアーキテクチャを**図 3** に示す．このアーキテクチャでは，Web サービスに必要な SOAP メッセージの処理を SOAP エンジンが担当し，WS-Security に必要な署名および暗号化の処理を WS-Security エンジンが担当している☆．通信プロトコルとして HTTP [20] を想定しているが，本実装は HTTP 以外のプロトコルにも同様に適用可能である．図 3 におけるメッセージ送信時の処理手順を以下に示す．

(1) アプリケーションが SOAP エンジンに必要なパラメータを渡す．
(2) SOAP エンジンが受け取ったパラメータを包括するリクエスト SOAP メッセージを UTF-8 バイト列として生成し，WS-Security エンジンに渡す．
(3) WS-Security エンジンが受け取ったメッセージの UTF-8 バイト列を SAX [7] パーザを用いて解析し，必要に応じてリクエスト SOAP メッセージに署名や暗号化を適用する．適用後も UTF-8 バイト列のまま保持する．
(4) HTTP 操作により保持しているリクエスト SOAP メッセージの UTF-8 バイト列をそのままサービスに送信する．

また，メッセージ受信時の処理手順を以下に示す．

(1) XML パーザがレスポンス SOAP メッセージを解析して SAX イベント列に分解する．

☆ WS-Security エンジンは，SOAP エンジンに対する一種のプラグインであり，不必要な場合には実装しなくてもよい．

表 1 WS-Security の各処理に必要な情報の一覧
Table 1 Necessary information for WS-Security.

処理	必要となる情報
署名・署名検証	署名（検証）対象に関する情報
署名のみ	署名値算出に関する情報
暗号化・復号化	暗号化（復号化）対象に関する情報
暗号化のみ	鍵の暗号化に関する情報

(2) WS-Security エンジンは受け取った SAX イベント列をもとに必要に応じてメッセージの署名を検証し，暗号を復号化する．復号化されたデータは WS-Security エンジン内で再帰的に解析される．WS-Security エンジンは処理した SAX イベント列のうち，SOAP エンジンが処理上必要とする SAX イベント列のみを順次 SOAP エンジンに流す．

(3) SOAP エンジンは受け取った SAX イベント列をもとにサービスの回答を抽出し，アプリケーションに渡す．

3.2 リクエスト SOAP メッセージの構築方法

本節ではリクエスト SOAP メッセージを構築する際の手順について詳細に述べる．

3.2.1 雛型を用いた SOAP メッセージの構築

SOAP エンジンは，アプリケーションから必要なパラメータ（呼び出すサービスとそれに必要な引数に関する情報）を受け取ると，あらかじめ用意された WS-Security 適用前の SOAP メッセージの雛型に可変長のパラメータを埋め込むようにして UTF-8 バイト列としてリクエスト SOAP メッセージを構築する．なお，著者らは SOAP メッセージの構築をストリーミング処理とは位置付けていない．

3.2.2 Web サービスセキュリティの設定

現状ではサービスに接続するために必要な WS-Security の要件を知るための標準技術に基づく実装が存在しないので，必要とされる情報を別途あらかじめ設定しておく必要がある．設定すべき情報は署名および暗号化の処理順序，**表 1** に示した各署名処理および暗号化処理に必要な情報である．各情報の詳細に関しては文献 3) を参照されたい．

3.2 節で述べる処理手順は汎用的なものであるが，説明を簡略化するために，以降次のようなシナリオを想定する．リクエスト SOAP メッセージには署名，暗号化の順で処理を施す．署名対象は<Body>要素全体とし，私有鍵で署名して公開鍵証明書をセキュリティトークンとして同包する．暗号化対象は<Body>要素内にある<para1>要素，<para2>要素とし，動的に生成する共有鍵で暗号化し，その共有鍵を受信者の公開鍵で暗号化して同包する．

```
BH[1]=...; LO[1]=...; // 署名用 — (0)
BH[2]=...; LO[2]=...; // 暗号化用 —(0)
Int m = 0;
Buffer buf = new Buffer(++m);
Stack bStack = new Stack(); // バッファ用
Stack eStack = new Stack(); // 暗号化遅延処理用

void invoke(byte[] mes) {
 Parser parser = ...;
 DefaultHandler handler = this;
 parser.parse(mes, handler); — (11)
}
void startElement(byte[] data) {
 if (<Body>要素, 署名対象, 暗号化対象の開始タグ) {
  Buffer nbuf = new Buffer(++m); —(1)(2)
  bStack.push(buf);
  buf.add(nbuf); buf = nbuf;
  if (署名対象) {
   LO[1].add (buf);  —(1)
  } else if (暗号化対象) {
   LO[2].add (buf);  —(2)
  }
 }
 buf.add(data); -- (3)
}
```

```
void characters(byte[] data) {
 buf.add(data); -- (3)
}
void endElement(byte data[]) {
 buf.add(data); -- (3)
 if (署名対象) {
  if (処理不可能？) エラーを返す —(9)
  // 署名処理を行い、BH[1]を埋める — (4)
  // 遅延した暗号化処理を行う —(10)
  buf.clear(); m--; -- (5)
  buf = bStack.pop();
 } else if (暗号化対象) {
  if (処理不可能？) エラーを返す —(9)
  if (先に署名？) eStack.push(buf); — (6)
  // 暗号化処理を行い、BH[2]を埋める— (7)
  buf.clear(); m--; -- (5)
  buf = bStack.pop();
 }
} else if (<Envelope>要素の終了タグ) {
 // SOAPメッセージの再構築を行う
 byte[] mes = reconMessage(BH[1], BH[2]) -- (8)
 // HTTP 操作を行う
 HTTP.write(mes); — (12)
}
}
```

図 4 メッセージ構築における WS-Security エンジンの擬似コード
Fig. 4 A sample code of WS-Security engine in request message construction.

3.2.3 WS-Security エンジンの処理手順

WS-Security エンジンの基本処理を**図 4** に擬似コードで表す．WS-Security エンジンは SOAP エンジンから受け取ったリクエスト SOAP メッセージ（UTF-8 バイト列）を上から順に解析（図 4 (11) に対応）して得られる SAX イベント列に基づき，下記の 2 つの処理を並行して行う．

- 署名対象および暗号化対象をバッファに格納する．
- 署名対象を署名し，暗号化対象を暗号化する．

これらの処理が無事終了すれば，SOAP ヘッダに署名検証，復号化に必要な情報を挿入すべく，メッセージの再構築を行う．

3.2.4 前 処 理

3.2.2 項で設定された署名および暗号化の処理順序に従って，処理に必要となる以下の 2 つのバッファを用意（図 4 (0) に対応）する．

- i 番目の署名処理あるいは暗号化処理に関して受信側で署名検証あるいは復号化するために必要な情報をバイト列（UTF-8）として格納するためのバッファ $BH_i(1 \leq i \leq m)$．BH_i はそのまま SOAP ヘッダに挿入される．m は署名処理および暗号化処理の総数である．なお，署名検証あるいは復号化するために必要な情報については文献 3) を参照されたい．
- i 番目の署名処理あるいは暗号化処理における署名対象あるいは暗号化対象を格納するバッファのリスト $LO_i(1 \leq i \leq m)$．LO_i にはその時点で入れ子になった対象の数だけバッファが含まれる．

3.2.5 署名対象および暗号化対象の格納

WS-Security エンジンは，リクエスト SOAP メッセージから<Body>要素，3.2.2 項で設定された署名対象および暗号化対象の開始タグを見つけだし，バッファ $B_x(1 \leq x \leq b)$ に格納していく．b はバッファの総

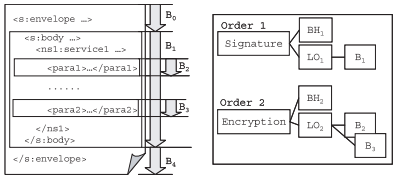


図 5 リクエスト SOAP メッセージの分割例
Fig. 5 An example of division of request SOAP message.

数とする．開始タグに対応する終了タグを見つけたところで，バッファ B_x への格納を止め，次項で述べる署名および暗号化の処理を行う．ただし，B_x への格納中に異なる署名対象および暗号化対象を見つけた場合にはバッファ B_{x+1} を用意し，B_{x+1} に格納していく．**図 5** にメッセージのバッファ分割の例を示す．図 5 では，署名対象（=<Body>要素）を見つけた時点で B_1 を署名用バッファリスト LO_1 に追加（図 4 (1) に対応）し，署名対象の中身を B_1 へとバッファリング（図 4 (3) に対応）していく．ただし，署名対象をバッファリングする際には XML 正規化を施したうえで署名対象の入れ子も考慮し，重複しない一意の名前空間接頭辞が使われるようにしたうえでバッファリングする必要がある．続いて，暗号化対象（=<para1>要素）を見つけた時点で B_2 を暗号化用バッファリスト LO_2 に追加（図 4 (2) に対応）する．B_2 へのバッファリングは<para1>要素の終了タグを見つけた時点で終了するが，<Body>要素への署名処理が先であるため，暗号化処理遅延スタックに B_2 を登録し，<para1>要素の暗号化処理を遅らせる（図 4 (6) に対応）必要がある．同様に暗号化対象（=<para2>要素）用のバッファ B_3 も暗号化用バッファリスト LO_2，暗号化処理遅延スタックに追加する．そして，<Body>要素の終了タグを見つけた時点で B_1 へのバッファリングを終了し，署名対象 B_1 に対して署名処理（図 4 (4) に対応）を行う．署名処理が終われば，処理を遅らせていた暗号化対象 B_2，B_3 を暗号化処理遅延スタックから取り出し，暗号化処理（図 4 (10) に対応）を行う．

3.2.6 署名処理および暗号化処理

WS-Security エンジンは，3.2.2 項で設定された処理順序に従って，署名処理および暗号化の処理を行う．**図 6** に署名処理および暗号化処理の概要を示す．本項では以降 k 番目の処理を行っているものとして各処理の詳細を記述する．

【署名処理】—図 4 (4) に対応

(1) $i(1 \leq i < k)$ 番目の暗号化処理のうちバッファリスト LO_i に対象を格納している処理があれば，処理を中断しエラーを返す（図 4 (9) に対応）．

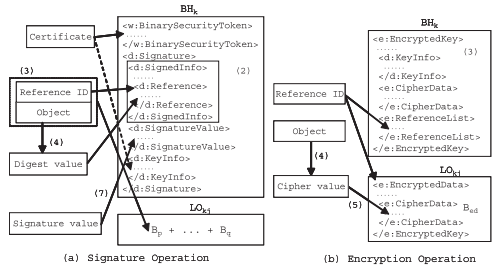


図 6 署名処理および暗号化処理の概要
Fig. 6 Signature and encryption operations.

(2) セキュリティトークンを含む<SecurityToken>要素と署名に関する設定情報を含んだ<Signature>要素の雛型を BH_k に格納する．すでに格納されている場合には何もしない．

(3) B_x に含まれる最初の開始タグに署名対象を参照するための識別子を付与し，その参照 URI を BH_k の所定の場所に格納する．

(4) 3.2.2 項で設定されたダイジェストアルゴリズムに基づいて，B_x で得られる UTF-8 バイト列からダイジェスト値を算出し，BH_k の所定の場所に格納する．

(5) $i(1 \leq i < k)$ 番目の署名処理のうちバッファリスト LO_i に対象を格納している処理がなければ，暗号化処理遅延スタックからバッファをすべて取り出し，順番に暗号化処理 (3)〜(6) を適用（図 4 (10) に対応）する．

(6) B_x を初期化（図 4 (5) に対応）する．

(7) すべての署名対象に対する処理を終えていれば，BH_k から<SignedInfo>要素を取り出し，署名値を算出し，BH_k の所定の場所に格納する．

【暗号化処理】—図 4 (7) に対応

(1) $i(1 \leq i < k)$ 番目の暗号化処理のうちバッファリスト LO_i に対象を格納している処理があれば，処理を中断しエラーを返す（図 4 (9) に対応）．

(2) $i(1 \leq i < k)$ 番目の署名処理のうちバッファリスト LO_i に対象を格納している処理があれば，暗号化処理遅延スタックに登録し，暗号化処理を遅らせる（図 4 (6) に対応）．

(3) 暗号化に関する設定情報を含んだ<EncryptedKey>要素の雛型を BH_k に格納する．すでに格納されている場合には何もしない．

(4) 3.2.2 項で設定されたデータ暗号化に利用する鍵，アルゴリズムに基づいて，B_x で得られる UTF-8 バイト列を暗号化する．

(5) データ暗号化によって得られたバイト列を<EncryptedData>要素で包み，<EncryptedData>

要素に暗号化対象を参照するための識別子を付与し，その参照 URI を BH_k の所定の場所に格納する．さらに，B_x の中身を<EncryptedData>要素の UTF-8 バイト列 B_{ed} で置き換える．

(6) B_x を初期化（図 4 (5) に対応）する．

3.2.7 リクエスト SOAP メッセージの再構築

署名および暗号化の処理が終わると，次の手順に従ってリクエスト SOAP メッセージを再構築する（図 4 (8) に対応）．

(1) B_0 の最後に<Header>要素と<Security>要素の開始タグを，B_1 の先頭に<Security>要素と<Header>要素の終了タグを挿入する．

(2) 各処理のために用意しておいたバッファを B_1 と B_2 の間に処理順の逆順（$BH_m, \ldots, BH_1$）に並ぶように挿入する．これはサーバ側の実装が WS-Security のヘッダを上から順に処理していくことを考慮したものである．

3.2.8 HTTP 操作

リクエスト SOAP メッセージの再構築が終わると，構築されたメッセージをネットワークに送るために HTTP 操作を行う（図 4 (12) に対応）．

3.2.9 メッセージ構築処理に関する考察

まず，著者らの実装で署名対象および暗号化対象の入れ子に対応していることを図 2 を例に確かめる．暗号化対象のみの場合，暗号化処理の順序により 2 通り考えられる．S2 の暗号化が先で S1 の暗号化が後の場合は問題なく処理できるが，S1 の暗号化が先で S2 の暗号化が後の場合は暗号化の性質上処理が不可能である．これは 3.2.6 項で示した暗号化処理 (1) に該当し，エラーを返す．署名対象のみの場合，署名処理の順序により 2 通り考えられるが，2.1 節で示した MProf (3) の制約により S1 および S2 のどちらが先に署名される場合でも問題なく処理できる．暗号化対象と署名対象が両方存在する場合でも，**表 2** に示すようにすべての場合に対応可能である．表 2 の一例を示すと，S1 の暗号化が先で，S2 の署名が後の場合は暗号化の性質上処理が不可能であり，エラーを返す．

また，再構築されたメッセージにおいて，セキュリティトークンは後方参照されるように挿入され，署名および暗号化対象は前方参照されるようになっており，2.1 節で示した MProf の制約に関してはすべて満たしている．

3.3 レスポンス SOAP メッセージの解析方法

本節ではレスポンス SOAP メッセージを解析する際の手順について詳細に述べる．

表 2 署名処理と暗号化処理の入れ子の分類
Table 2 Classification of signature and encryption nesting.

暗号化：対象	署名：対象	対処方法
先：S1	後：S2	エラーを返す 3.2.6 項署名処理 (1)
先：S2	後：S1	正常処理
後：S1	先：S2	正常処理
後：S2	先：S1	暗号化を遅らせる 3.2.6 項暗号化処理 (2)

```
LO[1]=...; // 復号化用 — (0)
LO[2]=...; // 署名検証用 —(0)
Int m = 0;
Buffer buf = null;
Stack bStack = new Stack(); // バッファ用
Stack eStack = new Stack();  // 復号化遅延処理用
SOAPEngine sE =...; // SOAPエンジン

void startElement(byte[] data) {
  if (in <Header>要素) { — (1)
    // 署名検証対象のダイジェスト値検証および
    // 復号化対象の復号化に必要な情報を抽出
  } else {in <Body>要素) {
    if (署名検証対象, 復号化対象の開始タグ) {
      Buffer nbuf = new Buffer(++m); —(2)(3)
      bStack.push(buf);
      buf.add(nbuf); buf = nbuf;
      if (署名対象) {
        LO[1].add (buf);   —(2
      } else if (暗号化対象) {
        LO[2].add (buf);   —(3)
      }
    }
    buf.add(data); — (4)
  }
  if (SOAPエンジンに渡す?) sE.startElement(); —(14)
}
void characters(byte[] data) {
  if (buf != null) buf.add(data); — (4)
  if (SOAPエンジンに渡す?) sE.characters(); —(14)
}
```

```
void endElement(byte data[]) {
  if (in <Header>要素) {
    if (<Signature>要素の終了タグ) {
      // 署名値の検証 — (5)
    } else if (<EncryptedKey>要素の終了タグ) {
      // データ復号化用鍵の復号化 —(6)
    }
  } else if (in <Body>要素) {
    buf.add(data); — (4)
    if (署名検証対象, 復号化対象の終了タグ) {
      if (署名検証対象) {
        // 署名検証対象のダイジェスト値検証 — (7)
        // 遅延した復号化処理を行う —(13)
        buf.clear(); m--; — (8)
        buf = bStack.pop();
      } else if (復号化対象) {
        if (先に署名検証?) eStack.push(buf);  — (9)
        // 復号化処理を行い、処理系を再帰呼出し
        byte[] ddata = decrypt(buf);
        Parser parser = ...;
        parser.parse(ddata, this); — (10)
        buf.clear(); m--; — (8)
        buf = bStack.pop();
      }
    }
  } else if (<Envelope>要素の終了タグ) {
    // 全署名検証・復号化対象の処理チェック — (11)
    // アプリケーションに戻り値を渡すことを許可 — (12)
  }
  if (SOAPエンジンに渡す?) sE.endElement(); —(14)
}
```

図 7 メッセージ解析における WS-Security エンジンの擬似コード
Fig. 7 A sample code of WS-Security engine in request message parsing.

3.3.1 WS-Security エンジンの処理手順

WS-Security エンジンの基本処理を**図 7** に擬似コードで表す．WS-Security エンジンは受け取った SAX イベント列からまず<Header>要素を解析（図 7 (1) に対応）しながら署名検証対象および復号化対象の処理に必要な情報だけを抜き出す．続いて，<Body>要素を解析しながら，下記の 2 つの処理を並行して行う．

- 署名検証対象および復号化対象をバッファに格納する．
- 署名検証対象のダイジェスト値を検証し，復号化対象を復号化する．

また，WS-Security エンジンは，受け取った SAX イベント列のうち，以下に該当するものだけを SOAP エンジンにも順次渡す（図 7 (14) に対応）．

- <Envelope>要素に関するイベント列．
- <Body>要素とその子要素全体に関するイベント列．ただし，<EncryptedData>要素とその子要素全体は復号化された後必要ないので除外する．

これらの処理がエラーを生じずに終了すれば，SOAP エンジンがアプリケーションに返り値を渡すことを許可（図 7 (12) に対応）する．

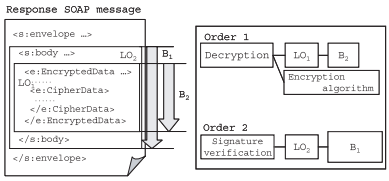


図 8 署名検証対象および復号化対象の格納例
Fig. 8 An example of the buffering of verifying and decrypting objects.

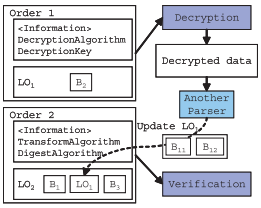


図 9 署名検証処理および復号化処理の概要
Fig. 9 Verification and decryption operations.

3.3.2 SOAP Header の解析

WS-Security エンジンは，<Header>要素に含まれる<Security>要素から，署名検証および復号化の処理の処理順序，各処理に必要な情報だけを抜き出して記憶する．表 1 に各処理で必要となる情報を示すが，<Header>要素の処理中に署名値の検証（図 7 (5) に対応）およびデータ復号化用鍵の復号化（図 7 (6) に対応）を行うため，これらの処理に必要となる情報を記憶しておく必要はない．なお，署名値の検証に失敗した場合には処理を中断し，エラーを返す．また，情報の抽出と並行して，各署名検証対象および復号化対象を格納するためのバッファリスト $LO_i(1 \leq i \leq m)$ も用意しておく．m は署名処理および暗号化の処理総数である．LO_i にはその時点で入れ子になった対象の数だけバッファが含まれる．

3.3 節で述べる処理手順は汎用的なものであるが，説明を簡略化するために，以降次のようなシナリオを想定する．レスポンス SOAP メッセージは復号化，署名検証の順で処理される．署名検証対象は<Body>要素全体とし，セキュリティトークンとして同包されている公開鍵を用いて署名を検証する．復号化対象は<Body>要素の子要素全体とし，私有鍵で復号化した共有鍵を用いて暗号化されたデータを復号化する，

3.3.3 署名検証対象および復号化対象の格納

WS-Security エンジンは，<Body>要素を解析しながら，3.3.2 項で得られた署名検証対象および復号化対象の開始タグを見つけ出し，バッファ $B_x(1 \leq x \leq b)$ に格納していく．b はバッファの総数とする．開始タグに対応する終了タグを見つけたところで，バッファ B_x への格納を止め，次節で述べる署名検証および復号化の処理を行う．ただし，B_x への格納中に異なる署名検証対象および復号化対象を見つけた場合にはバッファ B_{x+1} を用意し，B_{x+1} に格納していく．また，暗号化対象用のバッファには暗号化されたデータのみを格納するようにし，データ暗号化アルゴリズムは別途抜き出しておく．**図 8** に署名検証対象および暗号化対象をバッファに格納した例を示す．図 8 では，署名検証対象（=<Body>要素）を見つけた時点で B_1 を署名用バッファリスト LO_2 に追加（図 7 (2) に対応）し，署名検証対象の中身を B_1 へバッファリング（図 7 (4) に対応）していく．続いて，復号化対象（=<EncryptedData>要素）を見つけた時点で B_2 を暗号化用バッファリスト LO_1 に追加（図 7 (3) に対応）する．<EncryptedData>要素の終了タグを見つけた時点で B_2 へのバッファリングを終了し，暗号化データ B_2 の復号化処理（図 7 (10) に対応）を行う．<Body>要素の終了タグを見つけた時点で B_1 へのバッファリングを終了し，署名検証対象 B_1 のダイジェスト値検証（図 7 (7) に対応）を行う．

3.3.4 署名検証処理および復号化処理

WS-Security エンジンは，<Body>要素に含まれる署名検証対象および復号化対象をバッファに格納し終えると，3.3.2 項で得られた処理順序に従って，署名検証および復号化の処理を行う．**図 9** に署名検証処理および復号化処理の概要を示す．本項では以降 k 番目の処理を行っているものとして各処理の詳細を記述する．

【署名検証処理】—図 7 (7) に対応

(1) 3.3.2 項で得られるダイジェストアルゴリズムに基づいて，B_x で得られる UTF-8 バイト列からダイジェスト値を算出する．

(2) 3.3.2 項で得られるダイジェスト値と処理 (1) で算出した値を比較する．ダイジェスト値が正しい場合には「真（true）」を返し，異なる場合には「偽（false）」を返す．

(3) ダイジェスト値の検証が正しく終了すれば，現在の署名検証対象を処理したことを示すフラグを立てる．署名検証処理においてはメッセージ構造が変わらないため，次に示す復号化処理のように処理系を再帰的に呼び出す必要性はない．

(4) $i(1 \leq i < k)$ 番目の署名検証処理のうちバッファリスト LO_i に対象を格納している処理がなければ，復号化処理遅延スタックからバッファをすべて取り出し，順番に復号化処理 (2)〜(4) を適用

（図 7 (13) に対応）する．

(5) B_x を初期化（図 7 (8) に対応）する．

【復号化処理】—図 7 (10) に対応

(1) k 番目の処理よりも先に行われるべき $i(1 \leq i < k)$ 番目の署名処理のうち LO_i に対象を格納している処理があれば，復号化処理スタックにその署名処理が終了するまで復号化処理を遅らせる（図 7 (9) に対応）．

(2) <Header>要素から得られるデータ暗号化に利用する鍵，3.3.3 項で得られるデータ暗号化アルゴリズムに基づいて，B_x で得られる暗号化されたデータを復号化する．復号化されたデータもまた XML 文書の一部であるため，レスポンス SOAP メッセージを解析しているパーザから必要な情報（名前空間テーブル等）を引き継いだ新たな子パーザを動的に生成して処理させ，その出力となるイベント列を WS-Security エンジンが再帰的に処理できるようにする（図 7 (10) に対応）．

(3) 再帰処理も含めて処理が終了すれば，現在の復号化対象を処理したことを示すフラグを立てる．

(4) B_x を初期化（図 7 (8) に対応）する．

WS-Security エンジンは，<Body>要素の終了タグを読み込んだ段階で，<Header>要素から得られた署名検証対象および復号化対象の処理がすべて終了したかどうかを確認（図 7 (11) に対応）する．処理を終了していない対象が 1 つでもある場合には，処理を中断し，エラーを返す．

3.3.5 メッセージ解析処理に関する考察

著者らの実装で署名検証対象および復号化対象の入れ子に対応していることを確かめる．復号化対象が入れ子になっている場合，3.3.4 項の復号化処理 (2) で示したように，処理系を再帰的に呼び出すことで対応可能である．署名検証対象が入れ子になっている場合，2.1 節で示した MProf に準拠したメッセージであれば，メッセージ解析時に XML 正規化を必要としないため，3.3.4 項の署名検証処理が正常に働く．ただし，MProf に準拠していないメッセージも解析する場合には，メッセージ構築時と同様に署名検証対象のバッファリング時に XML 正規化を考慮しなければならない．署名検証対象および復号化対象が両方存在する場合でも，**表 3** に示すようにすべての場合に対応可能である．表 3 の一例を示すと，S1 の暗号化が先で，S2 の署名が後の場合は暗号化の性質上処理が不可能であり，エラーを返す．

表 3 署名処理と暗号化処理の入れ子の分類
Table 3 Classification of signature and encryption nesting.

暗号化：対象	署名：対象	対処方法
先：S1	後：S2	エラーを返す 3.2.6 項署名処理 (1)
先：S2	後：S1	正常処理
後：S1	先：S2	正常処理
後：S2	先：S1	暗号化を遅らせる 3.2.6 項暗号化処理 (2)

表 4 各エンジンのコードサイズ
Table 4 Jar file size of all engines.

エンジン	Streaming 版	DOM 版
SOAP	71 KB	85 KB
WS-Security	83 KB	110 KB

3.3.6 レスポンス SOAP メッセージからの返り値の抽出

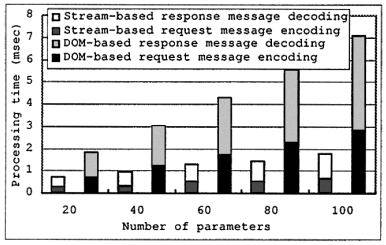
SOAP エンジンは，WS-Security 処理から渡された SAX イベント列だけを用いて，アプリケーションが必要とする返り値を抽出しておく．抽出した返り値は WS-Security エンジンから許可が出た段階でアプリケーションに返される．

4. ストリーミング処理による実装の性能評価

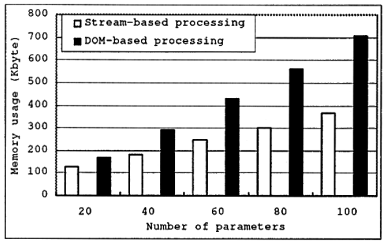
一般的な SAX パーザは内部でメッセージのバイト列（UTF-8）と文字列（UTF-16）の変換を行うが，文献 21) によると，これは時間のかかる処理である．そこで，著者らのプロトタイプでは SAX パーザのインタフェースを一部改良し，バイト列（UTF-8）のまま処理できるものを利用した．

本章では，筆者らが実装したストリーミング処理による SOAP エンジン（SE_{SAX} とする）および WS-Security エンジン（WE_{SAX} とする）のプロトタイプと WSTKMD [11] に搭載されている DOM に似た中間構造を用いた SOAP エンジン（SE_{DOM} とする）および WS-Security エンジン（WE_{DOM} とする）の処理時間やメモリ使用量を計測，比較する．SE_{DOM}，WE_{DOM} はともに J2ME 環境で動作するように，JSR172 [19] や 2.1 節で示した MProf の制約を満たす最小構成の実装になっており，最適化はなされている．ただし，SE_{DOM}，WE_{DOM} で利用している DOM に似たインタフェースでは Java 文字列（UTF-16）に変換されるため，メッセージをバイト列のまま処理することはできない．各エンジンの圧縮 jar ファイルのコードサイズを**表 4** に示す．

また，本章では市販の携帯電話実機上で著者らの SOAP エンジンが動作することも確認し，その処理時



(a) Relationship between number of parameters and processing time



(b) Relationship between number of parameters and memory usage

図 10 SOAP エンジンのストリーミング処理による実装と DOM を用いた実装の比較結果

Fig. 10 Performance comparison between two implementations of SOAP engine: streaming processing and DOM-based processing.

表 5 パラメータ数とメッセージサイズの関係

Table 5 Relationship between number of parameters and size of a message.

パラメータ数	1	20	100
リクエストメッセージ	572 B	1314 B	4435 B
レスポンスメッセージ	651 B	1184 B	3425 B

間を測定する．

4.1 SOAP エンジンの処理測定結果

SE_{SAX} と SE_{DOM} について，SOAP エンジンに渡されるパラメータ数を変化させながら，SOAP エンジンのメッセージ構築・解析にかかる処理時間（通信にかかる時間を除く）とメモリ使用量を比較した結果を**図 10** に示す．パラメータ数とメッセージサイズの関係を**表 5** に示す．表 5 からも分かるように，パラメータ数 100 の場合でメッセージサイズが 4 KB を超える．実験環境は IBM ThinkPad A31p☆（CPU：Pentium4 1.7 MHz，メモリ：756 MB，オペレーティングシステム：Windows2000）である．図 10 上部のグラフはパラメータ数と（X 軸）処理時間（Y 軸）の関係を，下部のグラフはパラメータ数（X 軸）と総メモリ使用量（Y 軸）の関係を示している．この結果から，SE_{SAX}（左棒グラフ）の方が SE_{DOM}（右棒グラフ）と比較して，処理速度が 2.5〜4.0 倍向上し，総メモリ使用量が 14〜42%減少していることが分かる．これらの結果には，DOM と SAX の処理の違いによる時間差とバイト列のまま処理することによる時間差が混在するが，現状では SE_{DOM} の実装を改良できないため両者を明確に分類できていない．これは今後の課題の 1 つである．また，パラメータ数が少ない場合に SE_{SAX} の方がメモリを必要とすることも分かっているが，その原因については現段階でつかめていない．この原因を究明することも今後の課題の 1 つである．

☆ “IBM” および “ThinkPad” は IBM Corporation の米国およびその他の国における商標．

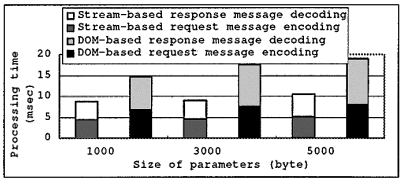


図 11 WS-Security エンジンのストリーミング処理による実装と DOM を用いた実装の比較結果

Fig. 11 Performance comparison between two implementations of WS-Security engine: streaming processing and DOM-based processing.

最後に，本実験においてパラメータ数を 20〜100 にした根拠を示しておく．

- SE_{SAX} と SE_{DOM} の処理時間や総メモリ使用量の差はパラメータ数が増加にともない大きくなることが容易に予想できる．
- 従量課金である携帯電話で 5 KB 以上のメッセージをやりとりすることは想定しがたいと考える．

4.2 WS-Security エンジンの処理測定結果

WE_{SAX} と WE_{DOM} について，SOAP エンジンのメッセージ構築・解析にかかる処理時間と WS-Security エンジンの署名・暗号化にかかる処理時間の合計を比較した結果を**図 11** に示す．実験環境は 4.1 節で示した環境と同一である．本実験では，SOAP エンジンに渡されるパラメータの数を 1 つだけとし，そのパラメータのバイト数を変化させた．WS-Security の設定に関しては，署名対象を<Body>要素全体，暗号化対象を<Body>要素の子要素全体とし，署名，暗号化の順で処理した．図 11 はパラメータのバイト数（X 軸）と処理時間（Y 軸）の関係を示している．この結果から，WE_{SAX}（左棒グラフ）の方が WE_{DOM}（右棒グラフ）と比較して，リクエスト SOAP メッセージの構築にかかる時間が 1.5 倍，レスポンス SOAP メッセージの解読にかかる時間が 1.8〜2.0 倍向上していることが分かる．WE_{SAX} と WE_{DOM} の処理時間の差はパラメータのバイト数が増加にともない大きくなることが容易に予想できる．

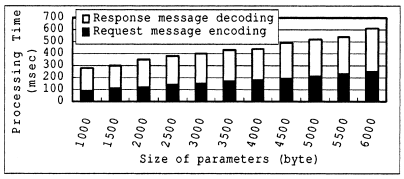


図 12 携帯電話上での SOAP エンジンの処理時間測定結果
Fig. 12 Performance of SOAP engine on real cellphone.

文献 6) では，短いメッセージを扱う場合にストリーミング実装の方が処理が遅くなることが報告されているが，WE_{SAX} を用いた今回の実験ではそのような現象は見られなかった．この理由としては，署名との併用による効果，DOM と SAX の処理の違いによる効果，メッセージをバイト列のまま扱うことによる効果が考えられるが，現状では明確ではない．今後暗号化のみの処理，署名のみの処理，暗号化，署名順の処理についても同様の実験を行うことで，WE_{SAX} が WE_{DOM} に比べて高速に動作する条件とその理由を考察したい．

4.3 携帯電話実機上での SOAP エンジンのプロトタイプと処理測定結果

著者らは市販の携帯電話上でも SOAP エンジンの稼動実験をしている．実験に使用した携帯電話は AU の A5303H である．基本性能については文献 22) を参照されたい．本実験は，4.1 節で示したストリーミング処理による実装に以下に示すような工夫を凝らして作成した 35 KB の圧縮 jar ファイルを Java アプリケーションとして携帯電話にダウンロードして行った．

- 引数として利用できる型を基本型 (boolen, int, String, ...) およびその一次元配列のみとした．
- クラスの総数や例外処理を極力減らした．
- 難読化 (Obfuscation) した．

携帯電話上での SOAP エンジンのメッセージ構築・解析にかかる処理時間（通信にかかる時間は除く）を測定するために，著者らは同一メッセージをサービスに反復送信し，初回を除いて，2 回目以降の処理時間の平均値を算出した．初回の測定時の処理時間を考慮に入れない理由は，処理に必要なライブラリをメモリ上にロードするのにかかる時間が実験結果に大きく影響すると判断したためである．

SOAP メッセージ中のパラメータの長さに対する処理時間の変化を測定した結果を**図 12** に示す．この結果から，著者らの実装が実際の CLDC 環境上でも動作することが確認できた．また，4.1 節と本節の実験結果から，大雑把にではあるが携帯電話実機上におけるはエミュレータ上における処理時間の約 350 倍かかることが分かる．SE_{SAX} 軽量版が 6 KB のメッセージを 350 ms で処理できるのに対して，仮に SE_{DOM} を携帯電話実機上で動作させた場合にはメッセージの処理に約 1500 ms かかることとなる．このことからも処理を高速化した著者らの実装の重要性が確認できたと考える．

4.4 携帯電話実機上での WS-Securtity エンジンのプロトタイプと処理測定結果

現在市販されている携帯電話上では WS-Security エンジンを稼動させることはできない．これは以下の理由によるものである．

- 携帯電話の持つセキュリティライブラリの API が公開されていない．
- ダウンロード可能なアプリケーションのサイズ制約が厳しい．

上記の問題を解決できれば，今後携帯電話の実機上で WS-Security エンジンの性能評価を行う予定である．

5. むすびに

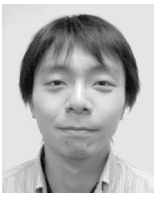
本論文では，計算機資源の限られた情報機器上でも Web サービスおよび WS-Security を実現するために，SAX を用いたストリーミング処理による実装を提案した．著者らの実装は XML 暗号化だけでなく，XML 電子署名も考慮した場合の問題を解決していることを示した．また，著者らが実装した SOAP エンジンおよび WS-Security エンジンのプロトタイプと WSTKMD に搭載されている SOAP エンジンおよび WS-Security エンジンの処理時間，メモリ使用量を比較することで，著者らの実装の有効性を示した．さらには，市販の CLDC に対応した携帯電話上でも著者らの実装が動作することを実際に確かめた．これらの結果からも，今後の各種情報機器上での Web サービスおよび Web サービスセキュリティの具現化が大きく期待できる．

参考文献

1) Berners-Lee, T., Fielding, R. and Masinter, L.: Extensible Markup Language (XML). http://www.w3.org/XML/
2) Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.J. and Nielsen, H.F.: SOAP (Simple Object Access Protocol) Version1.2. http://www.w3.org/TR/soap12/
3) Atkinson, B., Libera, G.D., Hada, S., Hondo, M., Baker, P.H., Klein, J., LaMacchia, B., Leach, P., Manferdelli, J., Maruyama, H., Nadalin, A., Nagaratnam, N., Pra-

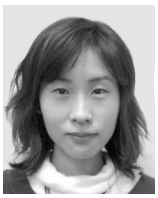
fullchandra, H., Shewchuk, J. and Simon, D.: Web Services Security. http://www-106.ibm.com/developerworks/webservices/library/ws-secure/
4) Web サービスのセキュリティとトランザクション管理，日経システム構築 2003 年 8 月号，pp.371–381 (2003).
5) Weiser, M.: Some Computer Science Problems in Ubiquitous Computing, *Comm. ACM*, Vol.36, No.7, pp.74–85 (1993).
6) Imamura, T., Clark, A. and Maruyama, H.: A Stream-based Implementation of XML Encryption, *Proc. ACM Workshop on XML Security*, pp.11–17 (2002).
7) Simple API for XML (SAX). http://www.saxproject.org/
8) Sun Microsystems, Java 2 Platform, Micro Edition (J2ME). http://java.sun.com/j2me
9) Ellis, J. and Young, M.: JSR30: J2ME Connected, Limited Device Configuration (CLDC) 1.0. http://jcp.org/en/jsr/detail?id=30
10) 益子由裕，並木美太郎：携帯電話向けの XML 処理用ミドルウェアの開発，マルチメディア，分散，協調とモバイルシンポジウム（DICOMO2003），No.9, pp.765–768 (2003).
11) Web Services ToolKit for Mobile Devices. http://www.alphaworks.ibm.com/tech/wstkmd/
12) Hors, A.L., Hegaret, P.L., Wood, L., Nicol, G., Robie, J., Champion, M. and Byrne, S.: Document Object Model (DOM). http://www.w3.org/DOM/
13) Eastlake, D., Reagle, J. and Solo, D.: XML-Signature Syntax and Processing. http://www.w3.org/TR/xmldsig-core/
14) Eastlake, D. and Reagle, J.: XML Encryption Syntax and Processing. http://www.w3.org/TR/xmlenc-core/
15) Nadalin, A.: SOAP Message Security: Minimalist Profile. http://lists.oasis-open.org/archives/wss/200303/pdf00002.pdf
16) Boyer, J., Eastlake, D. and Reagle, J.: Exclusive XML Canonicalization 1.0. http://www.w3.org/TR/xml-exc-c14n/
17) Yergeau, F.: UTF-8, a transformation format of ISO 10646, RFC2279. http://www.faqs.org/rfcs/rfc2279.html
18) Hoffman, P. and Yergeau, F.: UTF-16, an encoding of ISO 10646, RFC2781. http://www.faqs.org/rfcs/rfc2781.html
19) Ellis, J. and Young, M.: JSR 172: J2ME Web Services Specification. http://www.jcp.org/en/jsr/detail?id=172
20) Fielding, R.T., Gettys, J., Mogul, J., Nielsen, H.F., Berners-Lee, T., Masinter, L. and Leach, P.: Hypertext Transfer Protocol—HTTP1.1, RFC2616. http://www.faqs.org/rfcs/rfc2616.html
21) Bustamante, F.E., Eisenhauer, G., Schwan, K. and Widener, P.: Efficient Wire Formats for High Performance Computing, *Proc. 2000 Conference on Supercomputing* (2000).
22) HITACHI A5303H. http://www.hitachi.co.jp/Prod/vims/mobilephone/a5303h/

(平成 15 年 10 月 14 日受付)
(平成 16 年 4 月 5 日採録)



寺口 正義（正会員）

昭和 49 年生．平成 12 年大阪大学大学院基礎工学研究科情報数理系専攻修士課程修了．同年日本アイ・ビー・エム（株）入社．イメージ・メディア，Web サービスの研究開発に従事．



山口 裕美（正会員）

昭和 52 年生．平成 13 年お茶の水女子大学大学院人間文化研究科数理・情報科学専攻修士課程修了．同年日本アイ・ビー・エム（株）入社．情報可視化，Web サービスの研究開発に従事．

西海 聡子

昭和 45 年生．平成 8 年慶應義塾大学大学院理工学研究科物質科学専攻修士課程修了．同年日本アイ・ビー・エム（株）入社．現在同社東京基礎研究所主任研究員．フラットパネルディスプレイ，Web サービスの研究開発に従事．

伊藤 貴之（正会員）

昭和 43 年生．平成 4 年早稲田大学大学院理工学研究科修士課程修了．同年日本アイ・ビー・エム（株）入社．現在同社東京基礎研究所主任研究員．京都大学学術情報メディアセンター COE 研究員（客員助教授相当）兼任．博士（工学）．CAD，CAE，CG，情報可視化，Web サービスの研究開発に従事．