動的アプローチによる言語ベースの情報フロー制御

吉濱 佐知子[†] 工藤 道治[†] 小柳 和子[††]

機密性，完全性，可用性は情報セキュリティの基本的な理念であるが，特に複数の主体のインタラクションを通じた間接的な情報の流れにおいて機密性や完全性への要求を満たすことを，情報フロー制御（information flow control）と呼ぶ．しかし，セキュアOSのようにプロセス単位の情報フローの制御を行う場合，制御できる情報の粒度が大きく，アプリケーションレベルの機密保護要件を満たせず，また1つのプロセスの中で複数の機密度に属する情報が扱われる場合，プログラムの誤りによって不適切に情報が開示される危険がある．より粒度の細かい情報フロー制御を可能にするため，プログラミング言語のレベル情報フロー（Information Flow）を解析・制御する試みが行われている．しかし実用化に鑑みた場合，1) オブジェクト指向など疎結合な構成を持つソフトウェアで実行時の構成に起因する動的な振舞いを考慮できない，2) ハッシュテーブルなどの複雑なデータ構造や制御構造を通した情報フローの追跡が困難である，3) 言語の拡張やプログラムの改造が必要となるため既存のソフトウェア資産を再利用できない，などの阻害要因がある．本論文では，動的なアプローチを採用することにより，既存のソフトウェア資産に手を加えることなく情報フロー制御を行う方式を提案する．この方式では，プログラムの実行コードを書き換えることによって Inline Reference Monitor（IRM）を挿入し，実行時に動的に情報フローの追跡と制御を行う．バイトコード書き換え手法を使用することにより，ソフトウェアの改造が不要であり，実行環境に依存しないという利点がある．

Language-based Information Flow Control in Dynamic Approach

Sachiko Yoshihama,[†] Michiharu Kudoh[†] and Kazuko Oyanagi[††]

Information flow control refers to the concepts that satisfy confidentiality and integrity properties through indirect information propagation through interaction of entities. However, when information flow is controlled at the granularity of processes, such as in secure operating systems, the granularity of the control may be too coarse to meet requirements from application level compliance policies. In order to achieve fine-grained information flow control, *language-based information flow control* is receiving attention. The majority of past research focuses on static analysis of information flow, but when we consider the practical use of such technologies, there are difficulties; 1) Static analysis cannot cover the dynamic conditions of executing code or user input, 2) It is difficult to analyze complicated data structures and control flows of practically used object-oriented languages, and 3) When the language is extended with security types, existing software cannot be reused without adaptations. We propose a dynamic approach for information flow control that does not require modifications of existing code. In our approach, the system instruments the Java™ bytecode to insert an In-line Reference Monitor (IRM), and information is tracked and controlled by the IRM at the granularity of primitive data types. By using the bytecode rewriting technique, our approach works on existing programs without needing modification of source code, and it has no dependencies on specific Java virtual machines.

1. はじめに

機密性（confidentiality），完全性（integrity），可用性（availability）は情報セキュリティの基本的な理念であり，コンピューティング環境において情報を保護し，これらの性質を満たすことは長年の課題となっている．情報の機密性と完全性を保護するためには暗号技術やアクセス制御が多く利用され，特に主体の振舞いを通じて機密性や完全性への要求を満たすことを，情報フロー制御（information flow control）と呼ぶ．多くのセキュリティシステムにとって情報フロー制御は究極の目的であるが，従来の技術で完全に情報フローを制御することは困難である．

アクセス制御は一般的に，ある主体（subject）が対象（object）に対して一定の行動（action）をとることを許すかどうか，という一連のポリシを定義し，強

† 日本アイ・ビー・エム株式会社東京基礎研究所
Tokyo Research Laboratory, IBM Japan Corporation
†† 情報セキュリティ大学院大学
Institute of Information Security

制することによって，対象となる情報への読み書きを制御する．たとえば，機密情報の含まれるファイルを読み込む権限を，特定のユーザにのみ設定することが可能である．しかし，アクセス制御技術だけでは，情報のフローを制御することはできない．前述の例でいえば，機密情報の含まれるファイルへのアクセス権限を持った特権ユーザが，一般ユーザが読み込みできる他の公開ファイルへの書き込み権限をも持っていたとする．この場合，特権ユーザが，読み込んだ機密情報を公開ファイルに書き出すことにより情報が漏洩する可能性がある．

情報フロー制御とは，このように情報が主体によって間接的に伝播され，その伝播が多段階に及んでも，一定の機密性・完全性のポリシが保たれることを確実にするための技術である．

情報セキュリティ技術の歴史において，Bell-LaPadula[1] や Biba[2] など，情報フローを制御するためのポリシモデルが多く研究されている．また，これらのポリシモデルを実装するための技術も研究されており，特に軍用システムにおいては Bell-LaPadula[1] モデルを実装した Multi-Level Security（MLS）システムが実装され，機密レベルごとにシステムやネットワークの単位で強い分離が行われている．しかし，MLS システムは情報フローを厳密に制御するために，大きな実装上の制約を強いられる[3] ため，商用システムへの適用は容易ではない．

商用システムの要求を満たすためには，それぞれのコンピューティング・プラットフォーム上で強制アクセス制御（MAC: Mandatory Access Control）機能を実装し，情報フローを制御する必要がある．SELinux[4] ではオペレーティング・システム（OS）上にラベル・システムを実装し，プロセスや資源にセキュリティ・ラベルを付与し，OS 上のリファレンス・モニタが，ポリシによってラベル間のアクセス制御を行うことによって，情報フローを制御している．しかし，OS 上の情報フロー制御では，情報へのラベル付けの粒度に制限がある．1 つのプロセスの中で異なる機密度に属する情報が扱われる場合には，そのプロセスが正しくプログラミングされており，ポリシ違反となるような情報フローを起こさない必要がある．しかし実際には，プログラム開発者の不注意により，機密性の高い情報が適切に取り扱われずに漏洩する危険はつねに存在する．また，商用システムに求められる情報フローへの要望は多様化しており，たとえば非機密である情報の集合を個人情報として機密にする，または特定の情報の出力に際して監査ログを記録するなど，アプリケーションの処理内容によってより細かな制御が必要となる．

また，プロセス単位の情報フロー制御では，プロセス内の情報の伝播が不明であるために，ラベル進行（label creeping）問題によってポリシが不必要に厳密化するという問題がある（ラベル進行問題の詳細は 6.10 節に述べる）．

このような要求に応えるために，プログラミング言語のレベルで情報フローを制御する試みが行われている[5)~7)]．既存の研究の多くは，情報に関連付けられたセキュリティラベルをプログラミング言語上の型として静的解析を行うことにより，型理論に違反する情報フローを検知する．また，プログラミング言語を拡張し[8] セキュリティラベルを付与することにより，解析を容易にする．しかし，静的解析方式には，1) オブジェクト指向など疎結合な構成を持つソフトウェアで実行時の構成に起因する動的な振舞いを考慮できない，2) 実用的な言語におけるハッシュテーブルなどの複雑なデータ構造や制御構造を通した情報フローの追跡が困難である，3) 言語の拡張やプログラムの改造が必要となる場合既存のソフトウェア資産（実行環境，ライブラリ，開発ツールなど）を再利用できない，などの普及阻害要因が存在する．

このような背景をふまえ，昨今動的なアプローチによる情報フロー制御が注目を浴びている[9]．本論文では Java™ 言語に対して動的なアプローチを採用することにより，1) や 2) の問題に対応し，細粒度の情報フロー追跡を可能とするシステムを提案する．また，プログラムの実行コードを書き換えて情報フロー制御機能を追加することにより，ソースコードのないソフトウェアへの対応を可能とし，また実行環境，つまり Java 仮想マシン（JVM）に依存しないという利点がある．

以下の章では，まず 2 章で本論文で対象とする代表的なプログラム例を説明し，次に 3 章で言語ベースの情報フローの先行研究について述べる．4 章では，Java のバイトコード実行にともなって発生する情報フローについて説明する．5 章では，提案システムで用いる基本的な概念について説明し，6 章ではシステムのアーキテクチャと，その処理について詳細に述べる．7 章では，プロトタイプ実装について述べる．最後に 8 章でまとめと，今後の課題について述べる．

2. 適用シナリオ

Java による情報フロー制御が有効となるプログラム例を以下に示す．これは Web サーバ上のオンラインショップの簡便な例であり，HTTP リクエストか

らユーザ名と購入するアイテム名を受け取り，データベース上に登録されたユーザのクレジットカードを用いて購入処理を行う．購入処理（processPurchase メソッド）の内部ではクレジットカード番号の有効性をチェックし，有効期限が切れているなどの問題がある場合は，false を返して処理を中断する．ここで，ユーザより受信した HTTP リクエスト内の情報は非機密情報（ラベル LOW とする），データベースに格納されたクレジットカード番号は機密情報（ラベル HIGH）とする．なお，本メソッドが呼ばれる前にユーザ認証が完了しており，HTTP リクエスト中のユーザ名は信頼できるものと仮定する．

```
protected void doGet(HttpServletRequest request,
     HttpServletResponse response) throws ... {
  String user = request.getParameter("user");
  String item = request.getParameter("item");
  PrintWriter pw
   = new PrintWriter(response.getOutputStream());
  ...
  String credit = getCreditCardInfoFromDB(user);
  boolean b = processPurchase(user, item, credit);

  if(b){    // succeeds
    pw.println("Purchase Succeeded: <br/>");
    pw.println("Name: " + user + "<br/>");
    pw.println("Item: " + item + "<br/>");
    pw.println("Credit Card: " + credit );
  }else{    // failed
    printlog("Invalid credit card: " + credit);
  }
  ...
}
```

クレジットカードの機密性という観点で見た場合，このプログラム例には 2 つの問題がある．

(1) 購入処理が成功したとき，クレジットカード番号をユーザに対して送信している．通信経路の安全が確保されていたとしても，ユーザの端末でクレジットカード番号表示されることにより覗き見やブラウザのキャッシュを通じてデータが漏洩する危険がある．

(2) 購入処理が失敗したとき，クレジットカード番号をログファイルに出力している．システム運営上のログファイルへのアクセス権限が明確でないため，不必要な機密情報はログファイルに出力すべきではない．

本論文で提案するシステムの目的は，このようなプログラムを実行する環境に情報フロー制御機構を適用することにより，実行時に望ましくない情報フローの発生を防ぐものである．すなわち，上記の例では，クレジットカード番号を出力しようとした場合に，情報フローポリシ違反を検知して，処理を中断する．また，情報が適切にサニタイズされていると判断される場合には，その機密度を変更し，機密解除（declassification）を行う必要がある．たとえば上記の例では，クレジットカード番号の下 4 桁以外をマスクして “****-****-****-1234” のような形でユーザに提示することは慣例としてよく行われており，このような変換が行われたデータは，非機密情報であると判定できる必要がある．

3. 関連研究

きめの細かい情報フロー制御を行うために，昨今言語ベースの情報フロー制御・解析（Language-Based Information Flow Control/Analysis）が注目されている．本章では，言語ベースの情報フロー制御に関連する研究について考察する．言語ベースの情報フロー分析・制御は，静的に行う手法と，実行時に動的に行う手法に大別される．また，既存のプログラミング言語をそのまま使用するものと，既存言語を拡張しセキュリティ情報を付加するものがある．

本章では関連研究について，特に実用性の観点から，

(1) Java Reflection などにより実行時に決定されるプログラムコードへの対応が可能か，

(2) ユーザ名，ユーザ入力など実行時に決定する条件によるポリシに対応可能か，

(3) メソッド呼び出し，Java 例外などの複雑な制御構造に対応できるか，

(4) 配列や連想配列，ハッシュテーブルなどの複雑なデータ構造に対応できるか，

(5) プログラムの改造が必要となるか，

(6) 実行環境の改造が必要となるか，

の 6 つの点についての比較を行う（**表 1**）．

3.1 静的アプローチによる情報フロー解析

静的手法としては，セキュリティ情報をタイプ・システムとして言語に導入し，プログラムを解析することによってセキュリティ・タイプに違反する処理が行われていないことを検証するアプローチが主流である．

静的解析はプログラムの実行前に行うため，実行時

表 1 言語ベース情報フロー関連研究
Table 1 Language-based information flow related work.

観点	(1)	(2)	(3)	(4)	(5)	(6)
10)	×	×	×	×	不要	不要
11)	×	×	○	○	要	不要
7)	×	×	○	○	不要	不要
8)	×	○	○	○	要	不要
12)	○	○	○	○	不要	要
6)	×	×	○	×	要	不要
9)	○	○	○	○	不要	不要
提案方式	○	○	○	○	不要	不要

のオーバヘッドがなく，実行環境の改造が必要ない．また，可能性のある実行パスについて網羅的に分析することができる一方で，実行時の条件に依存するコードについて完全に分析することは難しい．たとえばJava Reflectionなどを利用したプラッガブルな実装や，スレッド間をまたがる情報フロー，ハッシュテーブルなど複雑なデータ構造を介した別名解析[13]は困難である．

Kobayashiら[10]は限定的なJavaバイトコード命令のサブセットを定義し，その上での情報フロー分析を行っている．ここでは既存のバイトコードを改造せずに解析可能であることを示しているが，対象とするプログラム構造は限定されている．

Bartheら[11]はセキュリティラベルにより拡張したプログラミング言語と，Javaのサブセットを定義した中間言語が等価であることを証明し，次にこの中間言語とバイトコードが等価であることを証明することによって，バイトコードがソースコードと同等の情報フローポリシを満たすことを証明している．ここではプログラミング言語をセキュリティラベルで拡張しているため，元のプログラムはこの言語で記述される必要がある．

Genaimら[7]は，より完全なJavaのバイトコードに対する静的フロー解析を行い，その実装を行った．メソッド呼び出しや制御構造にともなう暗黙的フロー，またJava例外にともなう情報フローなどにも対応している．この方式はプログラムのソースコードを必要としないという長所があるが，実行時の条件に依存する情報フローには対応できない．

Jif[8]ではJava言語を拡張しセキュリティラベルを記述することによって，きめ細かく広範囲な情報フロー制御を可能にしている．Jifで記述された言語は専用のコンパイラによって一般的JVM上で実行可能なバイトコードに変換されるため実行環境への依存はないが，ソフトウェアはJif言語で記述する必要がある．記述されたセキュリティラベルの多くはコンパイル時に静的解析されるが，ユーザ名など実行時の条件に依存するポリシは，実行時に動的に判定することが可能である．

3.2 動的アプローチによる情報フロー解析

動的手法は，実行されたパスのみが検証されるため，すべての実行パスについての網羅的な分析を行うことが難しい．また実行時オーバヘッドが発生するという欠点がある．一方で，実行時のプログラムの状態をすべて利用できるため，より正確な判断が可能である．Beresら[12]はオペレーティングシステムを改造し，各機械語命令の実行ごとに情報フローを追跡することによって，プログラム内での情報フローを追跡する仕組みを提案している．同様の方式をJavaに適用することも可能だが，その場合はJVM自体を改造する必要があるため実行環境の実装への依存が発生する．

Erlingssonら[14]はJavaのバイトコード書き換えによって実行コードにインライン・リファレンス・モニタ（IRM）を挿入し，Java2セキュリティ[15]と同等のアクセス制御を行う仕組みを提案している．

Haldarら[9]，Franz[16]はバイトコード書き換え技術を利用し，Javaプログラムに対する外部入力値にtaintフラグやセキュリティラベルを付与することによる情報フロー制御を提案している．この方式では，メソッド実行とフィールドアクセスの処理に対してインライン・リファレンス・モニタを挿入し，発生する情報フローを追跡する．Franz[16]の報告ではサポートされる粒度はJavaのオブジェクト単位と，オブジェクトのフィールドの単位であり，ローカル変数や，オペランドスタック上での演算，またJavaの例外を介した情報フローの追跡可能性や，明示的な機密解除方式については言及されていない．

本論文の提案はHaldarらの方式と同様にバイトコード書き換え技術を利用するが，より細かい粒度で情報フローを追跡する点が異なる．本論文の方式では整数や浮動小数点数を含むプリミティブな値や配列要素のそれぞれといった細かい単位で，Java仮想マシンのローカル変数やオペランドスタック操作を含めたほぼすべてのバイトコード命令によって発生する情報フローの追跡と制御が可能となっている．また，本論文では文献17)を拡張し，Javaバイトコード上での情報フローをより明確に定義するとともに，ポリシに基づいた明示的機密解除の方式を提案する．

4. Javaプログラム上の情報フロー

本章では，Javaプログラム実行の概要と，実行時に発生する情報フローについて説明する．

4.1 Javaプログラムの実行

Javaのアプリケーションが実行される場合，まずJava仮想マシン（JVM）が起動され，初期化コードからアプリケーションのエントリポイントとなるメソッドが呼び出される．Javaスタンドアロンアプリケーションの場合，エントリポイントは実行されるクラスのmain()メソッドである．プログラムの実行が進むにつれ，メソッドは別のメソッド，またはライブラリなどを呼び出す．各メソッドの実行は，正常復帰または例外の発生によって完了する．外部への出力または

入力は標準 API を介して行われる．最終的に，例外が発生するか，main() メソッドから復帰することによって，アプリケーションの実行が完了する．

Web アプリケーションの場合には，アプリケーションプログラムはサーバの提供するフレームワークの中で動作するため，あらかじめ定義されたメソッド（たとえば Servlet の場合は doGet メソッドなど）がエントリポイントとなる．

4.2 Java バイトコード上の情報フロー

Java 言語で記述されたプログラムは，仮想的な機械語である Java バイトコードに変換され，JVM 上で実行される[18)]．

提案方式では，ソースコードのないプログラムに対応するために，Java バイトコード単位での情報フローを，制御の対象とする．そのため本節では，Java バイトコード命令ごとに発生する情報フローを説明する．

JVM 上で扱うことのできるデータ種別にはプリミティブ型と参照型がある[18)]．プリミティブ型には byte（8 bit），short（16 bit），int（32 bit），long（64 bit），char（16 bit）という整数型と，float（32 bit），double（64 bit）という浮動小数点型がある．また参照型はオブジェクトに対する参照を表す型であり，32 bit で表現される．

Java バイトコードには約 200 の命令が存在する[18)]が，多くは異なるデータ型のために同様の命令が複数用意されていたり，また頻繁に使用されるローカル変数インデクスを含む命令が定義されていたりするため，意味的には重複がある．たとえばローカル変数に対するロード命令には ILOAD，LLOAD，FLOAD，DLOAD，ALOAD の 5 つがあり，それぞれ int 型，long 型，float 型，double 型，参照型に対応する．また，ILOAD 命令には ILOAD_0 から ILOAD_3 まで命令にローカル変数インデクス値を含むバリエーションがある．byte 型と short 型は int 型として扱われる．

表 2 に，Java バイトコード命令を機能ごとに大別し，それぞれの処理を示す．

この表から，多くのバイトコード命令によってオペランドスタックとローカル変数，またヒープ領域間での情報の伝播が行われることが分かる．また，分岐命令によって，制御構造に起因する暗黙的情報フローが発生する．これは 6.8 節で詳細に述べる．

4.3 クラスローダ

一般的な Web アプリケーションサーバでは，アプリケーションとフレームワークの Java クラス名前空間を分離するために，クラスローダを多段に配置する．個々のアプリケーションは専用のクラスローダ上にロードされ，フレームワークや共有ライブラリとは区別される．提案方式ではフレームワークやライブラリのコードは信頼されるものと定義し，信頼されないアプリケーションのコードを情報フロー制御の対象とする．

表 2 Java バイトコード命令
Table 2 Java bytecode instructions.

ニーモニック	意味
NOP	無処理
CONST	定数をオペランドスタック（OS）上にロードする
LDC	定数データを OS 上にロードする
LOAD	ローカル変数を OS 上にロードする
STORE	OS 上の値をローカル変数に格納する
ALOAD	配列要素を OS 上にロードする
ASTORE	OS 上の値を配列要素に格納する
POP	OS から値を取り出す
DUP	OS 上の値を複製し，OS 上に置く
ADD	OS 上の 2 つの値を加算し結果を OS 上に置く
SUB	OS 上の 2 つの値を減算し結果を OS 上に置く
MUL	OS 上の 2 つの値を乗算し結果を OS 上に置く
DIV	OS 上の 2 つの値を除算し結果を OS 上に置く
REM	OS 上の 2 つの値の剰余算し結果を OS 上に置く
NEG	OS 上の 1 つの値を符号反転し OS 上に置く
SH	左辺シフトもしくは右辺シフトを行う
AND	OS 上の 2 つの値の論理積を求める
OR	OS 上の 2 つの値の論理和を求める
INC	指定されたローカル変数に値を加算する
x2y	OS 上の値の型を x から y に変換する
CMP	OS 上の 2 つの値を比較し結果を OS 上に置く
IF*	2 つの値を比較し，結果により分岐する
GOTO	無条件に指定されたアドレスに分岐する
JSR/RET	Finally 項への分岐と復帰を行う
SWITCH	テーブルによる分岐を行う
RETURN	メソッドより復帰する
GETSTATIC	スタティックフィールドから値を取得する
PUTSTATIC	スタティックフィールドに値を設定する
GETFIELD	オブジェクトフィールドから値を取得する
PUTFIELD	オブジェクトフィールドに値を設定する
INVOKE	メソッドを実行する
NEW	オブジェクトを生成する
THROW	例外を発生する
CAST	型検査を行う
MONITOR	マルチスレッドの排他検出

5. Java 情報フロー制御の概要

プログラム実行による情報フローとは，ある外部環境 S からの入力がプログラム内を伝播して別の外部環境 D に出力されることで発生する．外部環境 S の情報が外部環境 D に通知されるのが望ましくない場合，このような情報フローは防止されるべきである．

本論文では，Java プログラム実行時の望ましくない情報フローを防止するために，1) プログラムに対する入力元・出力先となる外部環境 S と D とそれらのセキュリティラベルを定義し，2) ラベル間の情報フ

ローポリシを強制し，3) プログラム実行中の情報フロー伝播を追跡する仕組み，の 3 つを提供する．

5.1 入出力のラベル付けポリシ

Java プログラムへの基本的入出力には，コマンドライン引数，標準入力，標準出力，ファイルおよびネットワークへのアクセスなどがある．またこれらの入出力を包含した上位の概念として，ロギング，データベースアクセス，Servlet API での入出力，などが Application Programming Interface（API）経由で提供される．

いずれの場合もアプリケーションプログラムにとっての入出力は，API のメソッド呼び出しなど，ライブラリとのインタラクションによって発生すると考えられる．よって，ここでは，API を介する入出力先を “java:クラス名表現” といういう擬似 URI で表現する．また，ファイルやネットワークへのアクセスは同じ API であっても対象ファイル/ホスト・ポートによって意味合いが異なってくるため，アクセス対象を URL で表現する．

たとえば 2 章のプログラム例では，入力値として HTTP リクエストを非機密，データベースから読み出したクレジットカード番号を機密，出力としては HTTP レスポンスやログファイルへの出力を非機密と，ラベル付ける．

5.2 情報フローポリシ

提案方式はポリシ非依存であるため，情報フローポリシとしては，要件に従って Bell-LaPadula [1] や Biba [2] などの異なるポリシを採用することが可能である．以下の例では簡便のため，2 つのラベル HIGH，LOW 間の秘匿性を考慮したシンプルな Bell-LaPadula モデルに基づくポリシを採用する．つまり，同ラベル間と LOW から HIGH への情報フローは許されるが，HIGH から LOW へのすべての情報フローは許されない．なお明示的にラベルの付いていない変数やオブジェクトには NONE という仮想的なラベルが付いているものとして扱い，HIGH または LOW からの情報フローが生じた時点でそのラベルが伝播される．

5.3 ラベル合成

異なるラベルを持つ 2 つの値から別の値が導出されるとき，導出された値のラベルは，元になっている値のラベルを合成したものとならなければならない．ここでラベルの合成は，2 つのラベルを最低限満たす Least Upper Bound（LUB）[8] とする．たとえば $a + b = c$ であり，a が HIGH，b が LOW であるときは，c のラベルは HIGH となる必要がある．なぜなら，c を LOW とした場合，b と c の値を知った攻撃者は a を推測することが容易であるためである．

6. IRM による情報フロー制御機構

情報フロー制御のアーキテクチャを**図 1** に示す．

図 1 の下部は，JVM の持つ内部構造を表している．VM は実行中の各スレッドごとの実行中の状態を保持する構造を持ち，それぞれのスレッド t の実行 Frame を保持する JVM Stack（js^t）とプログラムカウンタ（pc^t）を持つ．メソッド呼び出しが行われるときに新しい Frame fr_i^t が作成され，js^t 上に push される．それぞれの fr_i^t はローカル変数テーブル lv_i^t とオペランドスタック os_i^t を持つ．また，オブジェクトの実体はヒープ領域に格納される．

アクセス制御モジュール（ACM）は Java によって実装されたクラスである．ACM は JVM の持つ内部構造に対応するデータ構造（$l(pc^t), l(fr_i^t), l(os_i^t), l(lv_i^t)$）を持ち，実際にプログラム中で操作される値の代わりに，値に関連付けられたセキュリティラベルを保持する．また，オブジェクトやそのフィールドに関連付けられたラベルを管理するための Object Label Table（OLT），配列要素のラベルための Array Label Table（ALT），そして各クラスのスタティックフィールドのための Class Label Table（CLT）を持つ．

提案方式では，情報フロー制御を行うために本来の Java バイトコードに挿入された命令群をインライン・リファレンス・モニタ（IRM）と呼ぶ．IRMWriter は Java バイトコード書き換えを行う Java クラスであり，バイトコード命令が実行されるごとに，JVM 内の状態を ACM の持つ状態に反映させるための IRM を，Java アプリケーションコード内に挿入する．IRMWriter によって，アプリケーション実行前に一括してコードを書き換えることも，実行時にクラスがロードされるご

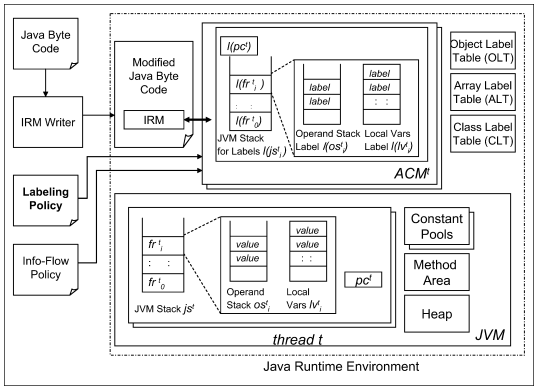


図 1 アーキテクチャ
Fig. 1 Architecture.

とに動的に書き換えることも可能である．

実行時に，プログラムに埋め込まれた IRM コードが ACM の機能を呼び出し，ACM と JVM の状態を同期させて，ACM が JVM で持つデータのラベルを管理するようにする．この機構により，JVM に手を加えることなく，JVM 上で演算に使用されているデータのセキュリティラベルを管理することができる．ACM は入出力ラベル付けポリシと，情報フローポリシを参照して，情報フローの伝播と制御を行う．IRM によって書き換えられたバイトコードはポリシ非依存であるため，一括してバイトコード書き換えを行った場合でも，実行時のポリシ変更は容易である．

以下に，バイトコード命令ごとの ACM の処理を説明する．

6.1 定数操作

CONST/LDC 命令によって定数をオペランドスタック os_i^t 上にロードする場合，定数には明示的なラベルは付与されていないと考えられるので，ACM はラベル NONE を $l(os_i^t)$ 上に push する（ただし暗黙的フロー検知のためにプログラムカウンタに関連付けられたラベル $l(pc^t)$ を合成するが，これについては後述する）．

6.2 ローカル変数操作

ローカル変数に対する操作は LOAD，STORE の 2 種類である．ある変数から別の変数への代入は，LOAD 命令と STORE 命令で表現できる．LOAD 命令では指定されたインデックスのローカル変数 lv_i^t の値が，os_i^t 上に push される．このとき ACM は，現在の frame $l(fr_i^t)$ 上のローカル変数のラベル $l(lv_i^t)$ を $l(os_i^t)$ に push する．STOER 命令は逆に，os_i^t から値を pop して lv_i^t に格納する．このとき ACM は，$l(os_i^t)$ の先頭にあるラベルを pop して，指定されたインデックスの $l(lv_i^t)$ に設定する．

6.3 オペランドスタック操作

POP 命令では JVM は os_i^t 上の先頭の値を取り除くので，ACM は $l(os_i^t)$ からラベルを 1 つ pop して取り除く．DUP 命令は JVM は os_i^t 内の先頭の値を複製して os_i^t に push するので，ACM は $l(os_i^t)$ に対して同じ操作を行う．

6.4 1 項演算

一項演算は，os_i^t 上の値を符号反転する NEG 命令，lv_i^t 上の値を加算する INC 命令である．演算結果はもとと同じ場所に置かれるので，ラベルの操作は必要ない（ただし後述の暗黙的フロー検知を除く）．

6.5 2 項演算

二項演算は四則演算（ADD，SUB，MUL，DIV），余剰（REM），シフト（SH），論理和（OR），論理積（AND），排他論理和（XOR），型変換，比較（CMP）である．二項演算命令では，JVM は os_i^t 上から 2 つの値を pop して演算を行い，結果を os_i^t 上に push する．このとき ACM は，$l(os_i^t)$ から 2 つのラベルを pop し，その 2 つのラベルを合成した結果を $l(os_i^t)$ 上に push する．

6.6 オブジェクトとクラス参照

各オブジェクトはそれ自体にラベルが関連付けられるが，オブジェクトの持つフィールド（オブジェクト変数）についても，オブジェクト自体と異なるラベルが関連付けられる可能性がある．そのため，提案方式ではこの 2 つを区別して管理することが可能になっている．

JVM ではオブジェクトはヒープ領域に格納され，オブジェクトフィールドへのアクセスは，PUTFIELD と GETFIELD という専用のバイトコード命令により，オペランドスタック os_i^t を介して行われる．

ACM はオブジェクトインスタンスに対応するテーブル OLT を持ち，個々のオブジェクトインスタンスと，それぞれのインスタンスのオブジェクトフィールドに格納されている値のラベルを保持する．OLT はオブジェクト参照により検索可能なテーブルである．PUTFIELD 命令が実行されると，JVM は os_i^t 上の値を pop してオブジェクトインスタンスのオブジェクトフィールドに値を設定する．ACM は OLT の当該オブジェクトのエントリに $l(os_i^t)$ 上のラベルを伝播させる．GETFIELD 命令は逆に，OLT から $l(os_i^t)$ 上にラベルを伝播する．

オブジェクトフィールドに明示的なラベルが関連付けられていない場合，オブジェクト自体のラベルをそのオブジェクトフィールドのラベルとして扱う．これは，入力ラベル付けポリシによって受信したオブジェクトにラベルが関連付けられている場合，そのオブジェクトの持つオブジェクトフィールドにもそのポリシが伝播されるべきと考えられるからである．逆に，オブジェクトフィールドは明示的なラベルを持つがオブジェクト自体がラベルを持たない場合，オブジェクトフィールドのラベルはオブジェクト自体には影響しない．そうした場合に，不必要な範囲にラベルが拡散するのを防ぐためである．

スタティックフィールドへの読み書きは PUTSTATIC 命令と GETSTATIC 命令により，OS を介して行われる．ACM はクラスラベルテーブル CLT によってスタティックフィールドのラベルを管理し，$l(os_i^t)$ との間でラベルを伝播させる．配列につ

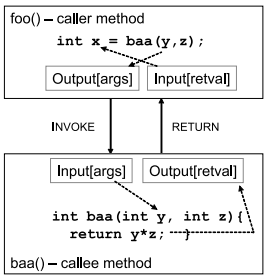


図 2 メソッド実行
Fig. 2 Method invocation.

いても同様に，JVM は配列要素にアクセスするための ALOAD 命令，ASTORE 命令が実行される際に，$l(os_i^t)$ と配列ラベルテーブル（ALT）との間でラベルを伝播させる．

6.7 メソッド実行

メソッド実行にともなう情報フローは，メソッド呼び出しの引数と，その復帰値，もしくは例外を介して発生する．メソッド foo() が baa() を呼び出す場合，foo() の視点からは，メソッドの実引数が出力であり，復帰値が入力となる（**図 2**）．逆に baa() の視点からは，メソッドの仮引数が入力であり，復帰値が出力となる．そこで，入出力ラベル付けポリシはこの 4 つを区別したラベル付けを可能にする．メソッドに対する入力が行われる場合，明示的なポリシが指定されていれば，入力値にそのラベルが関連付けられる．逆に，メソッドからの出力が行われる場合，出力値と出力先のラベルを比較し，情報フローポリシ違反を検出した場合には，情報フロー例外を発生して処理を中断する．たとえば，ラベル HIGH のデータを，ラベル LOW に関連付けられているメソッドの実引数に指定した場合に，情報フロー違反が発生する．

また，バイトコード書き換えによる IRM 挿入は，Java アプリケーションコードに対してのみ適用され，Java の標準ライブラリや Web サーバ自身のコードは除外される必要がある．たとえば Java 標準ライブラリに IRM を挿入した場合，IRM のコード自身で使用している標準ライブラリから IRM が再帰的に呼ばれることになり，正常な動作ができない．このため，呼び出し側（foo()），呼び出される側（baa()）のメソッドがともに IRM 適用されているとは限らず，1) foo(), baa() の両方が IRM 適用済み，2) foo() のみが IRM 適用済み，3) baa() のみが IRM 適用済み，4) 両者とも IRM 適用されていない，の 4 つの可能性がある．

1) の場合，メソッド間をまたがるラベルの伝播は，メソッド呼び出し前後の $l(fr_i^t)$ と $l(fr_{i+1}^t)$ の間でラベルを伝播させることで実現する．メソッドが実行されるときバイトコードは当該オブジェクトの参照と実引数を順に os_i^t に push して INVOKE 命令を実行する．次に JVM の内部で新しい Frame fr_{i+1}^t が生成され，JVM Stack に push される．JVM は os_i^t から実引数を POP し，新しい Frame のローカル変数 lv_{i+1}^t の 0 番目から順にコピーする．ACM はこの挙動にあわせ，呼び出し側 $l(fr_i^t)$ の $l(os_i^t)$ にオブジェクト参照と実引数のラベルを push し，INVOKE 命令の実行によって $l(fr_{i+1}^t)$ を生成し $l(js^t)$ に push して，この上の $l(lv_{i+1}^t)$ に前 Frame の $l(os_i^t)$ 上のラベルをコピーする．メソッドから復帰する場合は逆に，$l(os_{i+1}^t)$ 上の復帰値のラベルを $l(os_i^t)$ にコピーし，$l(fr_{i+1}^t)$ を $l(js^t)$ から pop する．また，入出力ラベル付けポリシによりメソッドの入出力に明示的なラベルが関連付けられている場合は，そのラベルを優先する．

2) の場合は INVOKE 命令実行時には ACM による処理が行われるが，RETURN 命令を実行するコードは IRM 適用されていないため，ACM が呼び出されない．そのため，復帰値のラベルについては以下のルールに基づいて呼び出し側メソッドで決定する：
i) 入出力ラベル付けポリシによって復帰値に明示的なラベルが指定されている場合，そのラベルを採用する．
ii) 復帰値に明示的なラベルが指定されていない場合，メソッド実行対象のオブジェクと実引数のラベルを合成したものを復帰値のラベルと推測する．

3) の場合は INVOKE 命令実行時に ACM による処理が行われないため，入出力ラベル付けポリシによって明示的な指定がないかぎり，メソッド baa() では入力ラベルが不明なものとして，引数にラベル NONE を関連付ける．

6.8 制御構造による暗黙的フロー

プログラム中で明示的に代入が行われなくても，プログラムの制御構造を通じて間接的に情報フローが生じることを，暗黙的フロー（implicit flow）と呼ぶ[8),12)]．

```
1: x = 1; // ラベル=HIGH とする
2: y = 0; // ラベル=LOW とする
3: if(x == 1){
4:   y = 1;
5: }
```

上記の例では，プログラム実行後に y の値から x が推測可能であるため，HIGH から LOW への情報の流出が起こっていると考えることができる．

このような情報フローをとらえるために，$l(pc^t)$ で，実行中のプログラムカウンタ pc^t のセキュリティラベルをトラックする．上記の例では 3 行目で if 文の条

件式を評価した段階で，$l(pc^t)$ を HIGH に変更する．さらに if 文の内部で値が代入された後，変数 y のラベルに，$l(pc^t)$ を伝播させる．

上記コードをコンパイルし，Java バイトコードに変換すると以下のようになる．

```
0:    iconst_1
1:    istore_1   // 変数#1 は x
2:    iconst_0
3:    istore_2   // 変数#2 は y
4:    iload_1
5:    iconst_1
6:    if_icmpne  #11
9:    iconst_1
10:   istore_2   // y に 1 を代入
11:   ...
```

6 行目の IF_ICMPNE によって，os_i^t 上の値が評価され，分岐が行われる．分岐先までのコード (9, 10) は，このとき os_i^t 上にある 2 つの値（変数 x と定数 1）のラベルの影響を受ける範囲である．

影響を受けるデータはすべて，PC のラベルを合成した新しいラベルを設定される必要がある．この例では，6 行目の IF_ICMPNE が実行されたときに，$l(pc^t)$ を変数 x と定数 1 のラベルの合成，つまり HIGH に設定する．9 行目と 10 行目で実行される処理のそれぞれに，$l(pc^t)$ が合成され，この例では変数 y にラベル HIGH が伝播される．分岐の終了点である 11 行目で $l(pc^t)$ は LOW に戻される．

同様に，ここまで説明してきた ACM の処理のそれぞれで，実際にはオペランドのラベルに加えて PC のラベルが合成される．たとえば定数をロードする場合，$l(os_i^t)$ には定数の持つラベルと $l(pc^t)$ を合成した値が push される．

ただし，厳密には if 文の内部を実行した場合もしない場合も暗黙的情報フローが発生する．つまり，上記コード実行後に y の値が 0 の場合には，x は 1 ではない，という推測が可能であるが，動的手法では if 文の内部が実行された場合にしか情報の流出を追跡できない点に注意が必要である．

6.9 例　　外

Java の例外は，JVM が発生する場合，ライブラリが発生する場合，アプリケーションプログラム中で THROW 命令によって明示的に発生させる場合がある．アプリケーションプログラム中で明示的に発生させる場合には，例外オブジェクトに設定したデータ（例外メッセージなど）と，実行時の $l(pc^t)$ より，例外オブジェクトのラベルが決定する．例外が捕捉されない場合は JVM 内部で多段階にわたって Frame が pop されるが，ACM との同期をとるために呼び出しパス上の各メソッドでデフォルトの例外ハンドラを定義し，捕捉した例外のラベルを記録してすぐに例外を throw する処理が必要になる．

JVM やライブラリが発生する例外は，IRM によって例外発生時の PC のラベルを検知することが可能だが，例外オブジェクト自身に含まれる情報についての完全な推測は難しい．たとえば，java.lang.ArithmeticException により 0 除算が通知される場合，その演算のオペランドが 0 であったということが推測される．しかしこの例外は JVM 自身から発生するので，IRM によってオペランドのラベルを例外に反映させることは難しい．

6.10 機 密 解 除

情報フロー制御を行う場合，ラベル進行（label creeping）問題を回避するために適切な機密解除（declassification）の機構が必要となる[19]．

Bell-LaPadula[1] モデルでは，機密情報から公開情報への情報フローを防ぐために，機密情報の読み込み権限を持つエンティティには公開情報への書き込みを許可しない．このようなシステムでは，情報の伝播を重ねるうちに，本来公開情報であるべき情報までもが徐々に機密情報とラベル付けされてしまう傾向がある．このような問題はラベル進行問題と呼ばれる[19]．OS レベルの強制アクセス制御では，プロセス内の処理をブラックボックスとして扱うために，このようなラベル進行問題を回避することは難しい．

プログラミング言語ベースの情報フロー制御であっても，プログラムを実行するに従って，従来ラベル LOW であったデータにもラベル HIGH が伝播していく傾向がある．特に，暗黙的フローを考慮するために $l(pc^t)$ をデータのラベルに合成した場合に，この傾向が顕著である．

しかし現実には，機密情報から生成されたデータがつねに機密情報とは限らない．たとえばクレジットカード番号は機密情報であっても，下 4 桁以外をマスクした “****-****-****-1234” のような文字列は公開情報として扱う，という慣習がある．また，パスワードは機密情報であっても，パスワードとチャレンジを連結した値の SHA-1 ハッシュ値や，パスワードとユーザ名による認証結果の boolean 値は公開情報である．

提案方式ではポリシによりメソッドの入出力単位で細かくセキュリティラベルを設定することにより，機密解除の要件を満たす．たとえば，クレジットカード番号の下 4 桁以外をマスクする String mask（String creditcard）というメソッドがあれば，入出力ラベル

付けポリシでこのメソッドの復帰値を LOW とラベル付けすることにより，プログラムを変更することなく機密解除を行い，ラベル進行問題を防ぐことができる．

7. プロトタイプ

提案システムのプロトタイプを，Apache Tomcat [20] 上に実装した．また，IRMWriter のプロトタイプは Apache Byte Code Engineering Library（BCEL）[21] で実装した．BCEL は Java バイトコードの変更・追加を容易に行うためのツールキットであり，オープンソースで公開されている．また Tomcat の Web アプリケーションクラスローダを改造し，アプリケーションクラスファイルのロード直前に IRMWriter を実行して動的にバイトコードを書き換えるようにした．なお今回作成したプロトタイプでは実装上の制限として，例外（Exception）による情報フロー，Finally 項，スレッド間の同期を用いた covert channel による情報フローはサポートしない．

図 3 に IRMWriter により IRM コードが挿入された状態のバイトコード例のダンプを示す．下線部分が挿入された IRM コードであり，ACM クラスを呼び出すことによって，情報フロー制御を行っている．

アプリケーションとしては，1) 整数・浮動小数点数の演算を行うプログラム，2) 2 章のプログラム例，の 2 つを用意し，複数メソッドにまたがって情報フローを追跡し，細かい粒度で制御できることを確認した．

以下に，2 章のプログラム例に適用される入出力ラベル付けポリシの例を示す．

```
<Policy>
<InputRule><Label>LOW</Label>
  <URI>java:foo.shop.PurchaseMask.doGet</URI>
  <Type>argument</Type></InputRule>
<InputRule><Label>LOW</Label>
  <URI>java:foo.shop.PurchaseMask.mask</URI>
  <Type>return</Type></InputRule>
<OutputRule>
  <Label>LOW</Label>
  <URI>java:foo.shop.PurchaseMask.printlog</URI>
  <Type>argument</Type>
<InputRule><Label>HIGH</Label>
  <URI>java:foo.shop.Purchase.
       getCreditCardInfoFromDB</URI>
  <Type>return</Type></InputRule>
</OutputRule>
```

このポリシでは，HTTP リクエスト，レスポンスをラベル LOW に，DB より取得されるクレジットカード番号にラベル HIGH，printlog() メソッドにより出力されるログに LOW のラベルを設定した．2 章のプログラム例にこのポリシを適用して実行した場合，情報フロー例外が発生することを確認した．

```
...
15:  iconst_0
16:  invokestaticcom.ibm.trl.acm.ACM.loadSingleVar (I)V
19:  aload_0
20:  invokestaticcom.ibm.trl.acm.ACM.pushSingleConst ()V
23:  iconst_0
24:  dup2
25:  invokestaticcom.ibm.trl.acm.ACM.loadSingleArray …
28:  aaload
29:  iconst_1
30:  invokestaticcom.ibm.trl.acm.ACM.storeSingleVar (I)V
33:  astore_1
34:  iconst_0
35:  invokestaticcom.ibm.trl.acm.ACM.loadSingleVar (I)V
38:  aload_0
39:  invokestaticcom.ibm.trl.acm.ACM.pushSingleConst ()V
42:  iconst_1
...
```

図 3 変換後のバイトコード例
Fig. 3 Example Java bytecode after instrumentation.

次に，クレジットカード番号の下 4 桁以外をマスクする mask() メソッドを導入し，ポリシによりこの復帰値をラベル LOW と定義した．2 章のプログラム例で出力されているクレジットカード番号をこの mask() メソッドにより変換することで，情報フロー例外を発生せずに正常にプログラム実行できることを確認した．

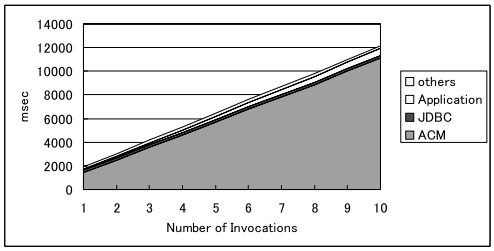
7.1 ポリシ定義に関する考察

提案方式では，入出力に関わる API 単位でのみポリシの定義が必要となるため，Jif [8] のようなセキュリティ型を持つ拡張言語に比べてポリシの定義は少なくて済む．上記の例では，2 章のプログラム例に適用される入出力ラベル付けポリシは入出力に関わる 4 項目である．実行時に扱われるデータのラベルが自動的に伝播されるため，外部入出力に関わらないメソッドについてはポリシを定義する必要がない．

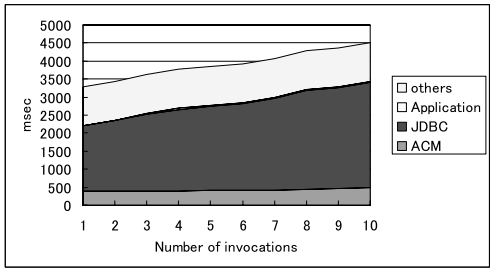
バイトコード書き換えによる IRM 挿入は，Java アプリケーションコードに対してのみ適用され，Java の標準ライブラリや Web サーバ自身のコードは除外される．IRM の挿入されていないメソッドが実行された場合には，呼び出し側で対象オブジェクトと実引数のラベルの合成を復帰値のラベルとして推測する．多くの API ではメソッドの復帰値はオブジェクト自身の値か引数の値によって決定するため，推測したラベルが有効な結果となる．しかし，オブジェクトが状態を持ち，過去のメソッド呼び出しによる入力値が保存されて後のメソッド呼び出しの復帰値の一部として出力されるような API については，あらかじめ入出力ラベルを定義する必要がある．ただし，標準的な API のラベルはあらかじめ定義しておくことにより，アプリケーション開発者の負荷を軽減させることは可能である．

7.2 パフォーマンス評価

IRM の挿入による実行時間の増加を，1) JDBC を用いずに数値演算のみ大量にを行うアプリケーション



(a) 数値計算Webアプリケーション実行時間計測



(b) JDBC使用Webアプリケーション実行時間計測

図 4 パフォーマンス計測
Fig. 4 Performance measurement.

(**図 4** (a))，2) 2 章のプログラム例を実装した JDBC 接続を行う Web アプリケーション（図 4 (b)）の 2 つを 10 回ずつ実行し，プロファイラを用いてパッケージごとの累積処理時間を計測した．なお，Web サーバ（Apache Tomcat）と DB（MySQL）は同一 PC 上で稼動させた．1) の場合，アプリケーション自体のクラスと ACM の実行時間を比較したところ，約 18〜24 倍の増加が認められた．2) の場合には，JDBC による DB 接続とクエリ処理が全体の処理時間が大部分を占めるため，ACM によるオーバヘッドはアプリケーション全体の 10%程度にとどまった．ただし，Web アプリケーションのクラスと ACM のみの実行時間を比較した場合は約 13〜18 倍の増加が認められた．なお IRMWriter によるコード書き換え処理はアプリケーションがロードされる最初の 1 回だけ行われるため，オーバヘッドのほとんどは ACM の処理によるものである．つまり，ACM によるオーバヘッドはアプリケーションクラスのサイズに依存するが，現実的な Web アプリケーションの多くは DB 接続を行うため，レスポンスタイムは実用的な範囲に収まるものと期待される．オーバヘッドは Haldar ら[9),16)] の方式と比べると大きいが，これは情報フロー追跡の粒度の差によるものである．

以上の結果から，アプリケーション自体の処理に対する ACM のオーバヘッドは大きいが，JDBC を使う一般的な構成の Web アプリケーションであれば許容しうる範囲のものと思われる．

また，IRM 挿入によるクラスファイルの変化を，約 2 KB から 22 KB までの異なる大きさのクラスファイルで計測したところ，約 2〜2.8 倍の増加が認められた．

8. まとめと今後の課題

バイトコードを書き換えて IRM を実装する方式は，
(1) 既存の Java コード上で情報フロー制御が可能，
(2) ソースコードやその改造が不要，
(3) JVM の実装に依存しない，
という長所がある．また本論文で提案した方式では，複数のメソッド呼び出しにわたって，オブジェクトだけでなく整数や浮動小数点を含む primitive 値や配列などの情報フローを細かい粒度で制御できるという利点がある．

一方で，提案方式を実運用環境で使用するにはいくつかの課題がある．まず，バイトコード命令単位での IRM では，情報フローを完全に把握するために JVM 上で実行される各命令ごとに対応する情報フロー制御処理を行う必要があるため，オーバヘッドが大きい．オーバヘッドを削減するためには，ACM による実行時チェック処理を最適化し，情報フロー制御処理を呼び出す頻度を低減させるなどの工夫が必要になると思われる．また，現在のプロトタイプではオブジェクトの参照と対応するラベルをテーブル上に格納しているため，ラベル管理の対象となるオブジェクトはガーベージコレクションされず使用メモリが増加するという問題がある．これらの問題があってもテスト環境で使用することには大きな支障はないが，テストの網羅性が重要になる．

また，動的手法のみでは暗黙的フローを完全に排除できないという問題があり，静的手法を組み合わせて使うことも考えられる．

次に，現在の入出力ラベル付けポリシ指定による機密解除（declassification）の仕組みは，プログラムの改造は必要ないが，ポリシ定義者にプログラムの構造と各メソッドの機能についての知識を要求する．また，人的誤りによる情報フロー違反の可能性を排除できない．ソースコードに対する知識のない状態で効率良く，かつ安全に機密解除を行う方式は今後の課題である．

謝辞 本研究の内容に関して一緒にご議論いただいた日本 IBM 東京基礎研究所および情報セキュリティ大学院大学の方々に感謝します．また，本研究は，経済産業省，新世代情報セキュリティ研究開発事業の研究として行われたものです．

参 考 文 献

1) Bell, D.E. and LaPadula, L.J.: Secure computer system: Unified exposition and multics interpretation, Technical Report MTR-2997 Rev. 1, MITRE, MITRE Corporation, Bedford, MA 01730 (Mar. 1976).
2) Biba, K.J.: Integrity considerations for secure computer systems, Technical Report MTR-3153, Mitre (June 1975).
3) Karger, P.A.: Multi-level security requirements for hypervisors, *The 21st Annual Computer Security Applications Conference*, December 5-9, 2005, Tucson, Arizona (2005).
4) Loscocco, P.A. and Smalley, S.D.: Meeting critical security objectives with security-enhanced linux, *Ottawa Linux Symposium* (2001).
5) Sabelfeld, A. and Myers, A.C.: Language-based information flow security, *IEEE Journal on Selected Areas in Communications*, Vol.21, No.1 (2003).
6) Banerjee, A. and Naumann, D.A.: Stack-based access control and secure information flow, *Journal of Functional Programming*, Vol.15, Issue 2, pp.131–177 (Mar. 2005).
7) Genaim, S. and Spoto, F.: Information flow analysis for java bytecode, *Verification, Model Checking and Abstract Interpretation* (*VMCAI05*), Cousot, R. (Ed.), volume LNCS 3385, Paris, France, January 2005, pp.346–362, Springer (2005).
8) Myers, A.C.: Jflow: Practical mostly-static information flow control, *Symposium on Principles of Programming Languages*, pp.228–241 (1999).
9) Haldar, V., Chandra, D. and Franz, M.: Dynamic taint propagation for java, *Annual Computer Security Applications Conference* (*ACSAC*) (2005).
10) Kobayashi, N. and Shirane, K.: Type-based information flow analysis for low-level languages, *APLAS 2002*, pp.2–21 (2002).
11) Barthe, G., Naumann, D.A. and Rezk, T.: Deriving an information flow checker and certifying compiler for java, *IEEE Symposyum on Security and Privacy* (2006).
12) Beres, Y. and Dalton, C.I.: Dynamic label binding at run-time, *New Security Paradigms Workshop* (2003).
13) Landi, W. and Ryder, B.G.: A safe approximate algorithm for interprocedural pointer aliasing, *Proc. Conference on Programming Language Design and Implementation* (*PLDI*), Vol.27, pp.235–248, ACM Press, New York, NY (1992).
14) Erlingsson, U. and Schneider, F.B.: Irm enforcement of java stack inspection, *IEEE Symposium on Security and Privacy*, pp.246–255 (2000).
15) Gong, L., Mueller, M., Prafullchandra, H. and Schemers, R.: Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2, *USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, pp.103–112 (1997).
16) Franz, M.: Moving trust out of application programs: A software architecture based on multi-level security virtual machines, Technical Report No.06-10, TR.06-10, University of California, Irvine (Aug. 2006).
17) Yoshihama, S., Kudoh, M. and Oyanagi, K.: Inforation flow control for java with inline reference monitors, *Computer Security Symposium* (*CSS2006*), Kyoto, Japan (Oct. 2006).
18) Lindholm, T. and Yellin, F.: *The Java Virtual Machine Specification*, Addison-Wesley (1999).
19) Li, P. and Zdancewic, S.: Downgrading policies and relaxed noninterference, *ACM SIGPLAN Notices*, Vol.40, No.1, pp.158–170 (2005).
20) Apache tomcat. http://tomcat.apache.org/
21) Apache byte code engineering library (bcel). http://jakarta.apache.org/bcel/

(平成 18 年 11 月 29 日受付)
(平成 19 年 6 月 5 日採録)

吉濱佐知子（正会員）

1993 年青山学院大学経済学部経済学科卒業．同年（株）セック入社．2001 年より IBM T.J. Watson 研究所勤務，2003 年より現在まで日本アイ・ビー・エム（株）東京基礎研究所勤務．2007 年情報セキュリティ大学院大学情報セキュリティ研究科修士課程修了．トラステッド・コンピューティング，情報フロー制御，Web アプリケーションセキュリティ等の情報セキュリティ分野の研究に従事．ACM 会員．

工藤　道治（正会員）

1988 年東京大学大学院工学系研究科修士課程修了，同年日本アイ・ビー・エム株式会社東京基礎研究所入社．情報セキュリティの研究・開発に携わる．主にアクセス制御・セキュリティポリシーの研究に従事．2001 年に米国標準化団体 OASIS において XACML 委員会を設立，2003 年 2 月に国際標準となる．2002 年東京大学より博士（工学）の学位授与．現在，東京基礎研究所セキュリティ&プライバシー・グループ担当．電子情報通信学会会員．

小柳　和子（正会員）

1973 年東京大学理学部化学科卒業．1978 年同大学院理学研究科博士課程修了．同年日立ソフトウェアエンジニアリング株式会社入社．2004 年情報セキュリティ大学院大学教授．理学博士．システムセキュリティの研究に従事．IEEE，電子情報通信学会各会員．