セキュリティ解析アルゴリズムの実現と オブジェクト指向言語への適用に関する一考察

横森 励士† 大畑 文明† 高田 喜朗‡ 関 浩之‡ 井上 克郎†‡

†大阪大学大学院 基礎工学研究科 情報数理系専攻
〒 560-8531 大阪府豊中市待兼山町 1-3
Phone: 06-6850-6571 Fax: 06-6850-6574
‡奈良先端科学技術大学院大学 情報科学研究科
〒 630-0101 奈良県生駒市高山町 8916-5
E-mail: {yokomori, oohata, inoue}@ics.es.osaka-u.ac.jp
{y-takata, seki } @is.aist-nara.ac.jp

あらまし クレジットカード番号等，第三者に知られてはならない情報を扱うプログラムや，不特定多数の人間が利用するシステムにおいては不適切な情報漏洩を防ぐことは重要な課題である．このような情報漏洩を検出する手法として，セキュリティ解析アルゴリズムが提案されている．これは，プログラム入力に対してセキュリティクラス（情報の重要度）を与えることで，プログラム中に現れる各出力に対してセキュリティクラスを導出する手法であり，情報フロー解析に基づいている．
本研究では，手続き型言語に対する実装を行なったセキュリティ解析ツールの説明 および オブジェクト指向言語に対する解析アルゴリズムの適用についての考察を行う．
キーワード セキュリティ解析，情報フロー解析，手続き型言語，オブジェクト指向型言語

Implementation of Security Analysis Algorithm and Application to OO Programs

Reishi Yokomori†, Fumiaki Ohata†,Yoshiaki Takata ‡,Hiroyuki Seki‡ and Katsuro Inoue†‡

†Graduate School of Engineering Science, Osaka University
1-3 Machikaneyama-cho, Toyonaka, Osaka 560-8531, Japan
Phone: +81-6-6850-6571 Fax: +81-6-6850-6574
‡Graduate School of Inforamtion Science,Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara 630-0101, Japan
E-mail: {yokomori, oohata, inoue}@ics.es.osaka-u.ac.jp
{y-takata, seki } @is.aist-nara.ac.jp

Abstract For a program which handles security information such as a credit card number, it is very important to prevent inappropriate information leak to outside of systems.
In order to guarantee that a given program never allow information leak, an information security analysis algorithm was proposed. Security analysis is based on information flow analysis, and provide the security class (degree of secret) of each output of the program, when the security class of each input is given. In this paper we introduce a security analysis tool implemented for a procedural language, and also discuss analysis methods for object-oriented languages.
Key words security analysis, information flow analysis, procedural language, object-oriented language

1 まえがき

クレジットカード番号等，第三者に知られてはならない情報を扱うプログラムにおいて不適切な情報漏洩を防ぐことは重要な課題である．Denning らは [1, 2] において，プログラムを静的に解析し不適切な情報漏洩を防ぐ手法を提案した．これらの手法は再帰手続きを考慮していなかったが，[8] により再帰を含む手続き型プログラムに対するセキュリティ解析アルゴリズムが提案されている．

しかし，これまでの研究は手法の提案だけでその実現については触れられていない．本研究では, [8] で提案されている手法を実装し，適用事例をまじえながらその有効性を検証する．実装においては，大域変数への対応，手続き間解析の効率化のため，[8] のアルゴリズムに変更を加えている．

また，現在のプログラム開発環境において，C などの手続き型言語だけでなく，Java[7]，C++[5] 等いわゆるオブジェクト指向言語の利用が高まっている．オブジェクト指向言語には，従来の手続き型言語にはないクラス，継承などの新しい概念が導入されており，既存のアルゴリズムを適用することは難しい．そこで，オブジェクト指向言語へのセキュリティ解析アルゴリズムの適用に際し起こる問題点を考察し，その対処方法について述べる．

以降，2. ではセキュリティ解析について述べる．3. でセキュリティ解析アルゴリズムの実装について述べ，4. で作成したツールの概要および適用事例を紹介する．5. でオブジェクト指向プログラムへのセキュリティ解析アルゴリズムの適用について考察し，最後に 6. でまとめと今後の課題について述べる．

2 セキュリティ解析

情報漏洩を防ぐ手段として，*Mandatory Access Control* と呼ばれる次のようなアクセス制御法がよく用いられる [4]: 「各データに対してその機密度を表すセキュリティクラス（*Security Class*，以下，SC と省略する）を割り当てる．データ d の SC を $SC(d)$ と表す．同様に，ユーザ（プロセス）に対し，どの程度のデータまでアクセスできるかを表すクリアランス（*Clearance*）を割り当てる．ユーザ u のクリアランスを $clear(u)$ と表す．$clear(u) \geq SC(d)$ のときかつそのときのみ，ユーザ u はデータ d を読むことができる.」

しかし，このアクセス制御法の場合，クリアランスが $SC(d)$ 以上のユーザプログラムがデータ d を読み込み，それを，故意にまたは過失によって，クリアランスが $SC(d)$ より小さいユーザにもアクセスできる記憶域に書き出してしまうと，不適切な情報漏洩が生じる．

Denning らは，このような情報漏洩を防ぐためにプログラムを静的に解析する手法を提案した [1, 2]. この手法では，まず，プログラムの入力となる値や変数に対して SC を，ファイルなどの出力域に対してクリアランスを設定する．つぎに，プログラム中で利用される変数間に存在するデータ授受関係を表す情報フロー（*Information Flow*）に基づき，不適切な情報漏洩の検出を行う．不適切な情報漏洩を引き起こす情報フローとしては，

- 低い SC を持つ変数への代入文において，高い SC を持つ変数が参照される
- 低いクリアランスを持つ出力文において，高い SC を持つ変数が参照される

がある．また [3] では，[2] の手法を理論的に再検討し解析手法の一般化を試みている．

しかし，これらの手法では再帰手続きや大域変数を考慮していないこともあり，解析対象となるプログラムが単純な構造のものに限られていた．また，関数呼び出し文に対する解析の際，戻り値の判定は実引数全体に対してのみで，戻り値の計算に実際にどの引数の値が利用されているかを考慮していないなど，不正確な面もあった．

そこで，[8] によって，再帰を含む手続き型プログラムに対する情報フロー解析アルゴリズムが提案されている．この方法では，解析対象プログラム中すべての実行文について，その文の実行前後の各変数の SC 間で成り立つべき再帰的な関係式を定義する．この関係式に基づき，プログラムの各手続きの実行結果の SC を解析する．プログラムの main 関数の仮引数が $x_1, \ldots, x_i$，入力ファイルが $infile_1, \ldots, infile_j$，main 関数の戻り値が y_1，出力ファイルが $outfile_1, \ldots, outfile_k$ であるとき，実引数と入力ファイルに対応する i + j 個の SC の組が与えられると，それらを元に上記の関係式を同時に満たす最小解が繰り返し計算により求められる．そして，この解が戻り値と出力ファイルに対応する 1 + k 個の SC となる．

セキュリティ解析には，目的に応じて

- プログラムの入力値の SC から出力値の SC を求める [8]
- 入力値の SC と出力域のクリアランスとの矛盾

表 1: 解析対象の BNF 表記

```
型              = 標準型 | 配列型.
標準型          = “integer” | “boolean” | “char”
配列型          = “array”[ ] “of” 標準型 .

複合文          = “begin” 文の並び “end”.
文              = 基本文 |
                  “if” 式 “then” 限定文
                  “else” 文 |
                  “if” 式 “then” 文 |
                  “while” 式 “do” 文.
限定文          = 基本文 |
                  “if” 式 “then” 限定文 “else” 限定文 |
                  “while” 式 “do” 限定文.
基本文          = 代入文 | 手続き呼出文 |
                  入出力文 | 複合文 | 空文.
代入文          = 左辺 “:=” 式.
入力文          = “readln” [“(” 変数の並び “)”]|
出力文          = “writeln” [“(” 出力指定の並び “)”].
手続き呼出文= 手続き名 [“(” 式の並び “)”].
関数呼び出し= 関数名 [“(” 式の並び “)”].
```

を検出する [2]

の 2 つの手法があるが，本稿では前者に着目する．

3 セキュリティ解析アルゴリズムの実装

我々は [8] の提案を受け，セキュリティ解析ツールのプロトタイプ実装を行なった．本節では，実装の方針および解析の手順について述べる．ツールの詳細および適用事例は 4. で述べる．

3.1 実装の方針

対象言語は Pascal のサブセットであり，表 1 にその BNF 表記の一部を示す．また，実装においては以下の点で [8] と異なる．

- 大域変数への対応
  詳細は後述する．
- 手続き間解析の効率化
  [8] では，手続きを解析する際，可能性のあるすべての引数の SC の組み合せを考慮し，それぞれの手続き内解析の結果を保持させている．
  しかし，実利用においては引数の SC の最小上界のみ考慮すればよく，実装ではそのようにしている．
- SC の簡単化，入出力ファイルの単一化
  直観的な理解を容易にするため，SC は $\{high, low\}$，入力ファイル，出力ファイルは，それぞれ標準入力，標準出力に限定している．

3.2 解析の手順

解析は以下の 2 つのフェーズで構成されている．

Phase 1: 前提条件の入力
前提条件は以下のプログラム文で設定できる．
入力文: 入力文で読み込まれる値の SC
手続き（関数）宣言部: 仮引数の SC
また，大域変数および局所変数の SC の初期値は low とする．

Phase 2: 情報フロー解析
前提条件を元に情報フローを計算しながら，プログラム中の各出力文における SC を求める．
解析終了後，SC の高い出力文を表示する．

以降，**Phase 2:** 情報フロー解析に関して，大域変数の解析，手続き内解析，手続き間解析に分けそれぞれ述べる．

大域変数の解析

本実装においては，大域変数を

- 手続き呼び出し先に対する，仮想的な引数
- 手続き呼び出し元に対する，仮想的な戻り値

として扱うことで，その解析を可能にしている．

しかし，すべての手続き呼び出し文（手続き）に対し，大域変数の数だけの引数（戻り値）を用意するのは効率が悪い．すべての大域変数が各手続きで使われるとは限らないためである．

そこで，前もって各手続きで直接もしくは間接的に定義，参照される大域変数を調べ，必要なだけの仮想的な引数（戻り値）を用意すればよい．手続き P で直接的に定義，参照される大域変数とは，P 内で扱われる大域変数を指す．手続き P で間接的に定義，参照される大域変数とは，P を起点とする手続き呼び出し経路上に存在する，P 以外の手続きで扱われる大域変数を指す．

手続き内解析

はじめに，手続き内で利用される可能性のある大域変数，局所変数，仮引数をもとに，セキュリティクラス集合（*Security Class Set*，以降，SCset と省略する）を用意する．その要素は (変数, SC) の組である．また，各変数の SC の初期値は以下の通りである．

局所変数: low
仮引数: 対応する手続き呼び出し文の実引数の SC の上界
大域変数: 対応する手続き呼び出し文の直前での大域変数の SC の上界

その後，手続き内の先頭の文からプログラムの実行順に従い解析を行う．SCset の計算は図 1 に基づき，手続きの先頭の文を P_{start} とすると，ALGORITHM(P_{start}, $\emptyset$) から始まる．また，文 s の解析開始時点での SCset を SCset(s)，文 s の解析終

了時点での SCset を SCset(s') と表す．アルゴリズムは文 s の種類に応じて定義され，それぞれ

s の解析時の内部処理
s の解析終了時の SCset および 出力文の持つ SC

の形式で記述している．図 2 には図 1 で利用される要素を示している．

手続き間解析

手続き型プログラムは複数の手続きから構成されており，手続き呼び出し経路は複数存在するのが一般的である．また，再帰経路の存在も避けられない．そのため，ある手続き P の解析結果が，P の呼び出し元手続き P' の解析結果に（引数や大域変数を介して）影響を与える可能性があり，すべての手続きの解析結果が安定するまで，手続き呼び出し経路上に存在する手続きを繰り返し解析する必要がある．そのため，手続きをまたぐ情報フロー解析では，以下のようなものを用意し解析を行う．SCset $\mathcal{C}$，$\mathcal{S}$ はすべての手続きに 1 つずつ存在する．

解析リスト:

手続き呼び出し経路に基づく，手続きの解析順リストである．解析リストは逐次更新され，空になるとプログラム全体の解析は終了する．具体的な更新アルゴリズムは [9] で述べられている．

手続き開始時点での **SCset $\mathcal{C}$**:

ある手続き呼び出し文 s があったとき，対応する手続き P を解析する際に，P が保持している SCset $\mathcal{C}$ と s により渡される SCset $\mathcal{C}'$ の最小上界をとる．その結果，$\mathcal{C}$ より高ければ $\mathcal{C}$ をその値で再定義し P の解析を行う．一方，$\mathcal{C}$ と等価であれば P の解析は行わない．

手続き終了時点での **SCset $\mathcal{S}$**:

手続き P の解析後，P が既に保持している $\mathcal{S}$ と解析終了時点での SCset $\mathcal{S}'$ の最小上界をとる．その結果，$\mathcal{S}$ より高ければ，$\mathcal{S}$ をその値で再定義し，P を呼び出すすべての手続きを解析リストに再登録する．一方，$\mathcal{S}$ と等価であれば何もしない．

解析例

例として，図 3 の関数 f に対する解析を考える．大域変数，局所変数，引数から

SCset(8) := {(a, *low*), (x, *low*), (y, *low*)}

を定義し．先頭の文（8 行目）から解析を行なう．図 1 より，

(代入文)

cl := { c | x∈Ref(s)∧(x, c)∈SCset } ∪ *imp*;
$kill$:= { (x, c) | x∈Def(s)∧(x, c)∈SCset };
gen := { (x, $\sqcup_{c\in cl} c$) | x∈Def(s)};
SCset := SCset − $kill$ ∪ gen

(入力文)

$kill$:= { (x, c) | x∈Def(s)∧(x, c)∈SCset };
gen := SCset_{input}(s)
(* { x | x∈SCset_{input}(s) } = Def(s) *)
SCset := SCset − $kill$ ∪ gen

(出力文)

cl := { c | x∈Ref(s)∧(x, c)∈SCset } ∪ *imp*
SC_{output} := $\sqcup_{c\in cl} c$; SCset := SCset

(分岐文)**(if E then $\mathbf{B}_{then}$ else $\mathbf{B}_{else}$)**

cl := { c | x∈Ref(E)∧(x, c)∈SCset } ∪ *imp*;
SCset_{pre} := SCset;
Algorithm(B_{then}, $\sqcup_{c\in cl} c$); SCset_{then} := SCset;
SCset := SCset_{pre};
Algorithm(B_{else}, $\sqcup_{c\in cl} c$); SCset_{else} := SCset;
SCset := unite(SCset_{then}, SCset_{else})

(繰り返し文)**(while E do B)**

SCset_{pre} := ∅;
while SCset <> SCset_{pre} **begin**
  cl := { c | x∈Ref(E)∧(x, c)∈SCset } ∪ *imp*;
  Algorithm(B, $\sqcup_{c\in cl} c$);
  SCset := unite(SCset, SCset_{pre})
end
SCset := SCset_{pre}

(ブロック文)**(being $\mathbf{B}_1$; ... $\mathbf{B}_n$; end)**

Algorithm(B_1, $\sqcup_{c\in cl} c$);
...
Algorithm(B_n, $\sqcup_{c\in cl} c$)
SCset := SCset

(手続き呼び出し文)

呼び出す手続きを P とする.
SCset_{next} := ∅
for i := 0 **to** |$\mathrm{s}_{actuals}$| **begin**
  cl := { c | ($\mathrm{s}_{actuals}$[i], c)∈SCset };
  SCset_{next} := SCset_{next} ∪ { ($P_{formals}$[i], cl) };
end;
foreach x∈Ref'(P) **begin**
  SCset_{next} := SCset_{next} ∪
    { (x, c) | (x, c)∈SCset }
end;
SCset := SCset_{next};
手続き P 内の解析;
$kill$:= ∅;
for i := 0 **to** |$\mathrm{s}_{actuals}$| **begin**
  $kill$:= $kill$ ∪
    { ($P_{formals}$[i], c) | ($P_{formals}$[i], c)∈SCset }
end
SCset := SCset − $kill$

図 1: Algorithm(s, *imp*)

Ref(s): 文 s で参照される変数の集合
Def(s): 文 s で定義される変数の集合
Ref'(P): 手続き P で参照される大域変数の集合
Def'(P): 手続き P で定義される大域変数の集合
s$_{actuals}$**:** 手続き呼び出し文 s の実引数の集合
$P_{formals}$**:** 手続き P の仮引数の集合
SCset: 解析時点でのセキュリティクラス集合
SCset$_{input}$**:** 入力文 s において設定される，(変数, SC) を要素とする集合
SC$_{output}$**(s):** 出力文 s が持つ SC
$\sqcup$**:** 最小上界を求める演算子
unite(A, B): セキュリティクラス集合である A と B を一つにまとめる．各変数の SC は，A，B においてその変数の SC の最小上界とする．

図 2: アルゴリズムの要素

```
1:  program sample;
2:  var a : integer;
3:  ...
4:  function f(x : integer) : integer;
5:  (* 解析時の条件は a,x ← low と仮定 *)
6:  var y : integer;
7:  begin
8:    readln(y); (* ← high *)
9:    if a > 0 then
10:   begin
11:     a := y + 1;
12:     y := x − 1;
13:   end;
14:
15:   writeln(y);
16:   writeln(x);
17:
18:   f := y;
19:   end
20: ...
21: end.
```

図 3: 手続き内解析の例

$kill := \{(y, low)\}$; $gen := \{(y, high)\}$
SCset(8') := SCset(8) $- kill \cup gen$
$:= \{(a, low), (x, low), (y, high)\}$
を求め，これを SCset(9) として次の文（9 行目）の解析を行なう．以下同様に最後の文（18 行目）まで解析していくと，次の結果が得られる．
SCset(18') := $\{(a, high), (x, low), (y, high), (f, high)\}$

4 セキュリティ解析ツールとその適用

本節では，セキュリティ解析アルゴリズムを実現したセキュリティ解析ツールについて説明し，具体的な適用事例を取り上げその有効性を検証する．ツールの実現は，我々が開発したスライスツール

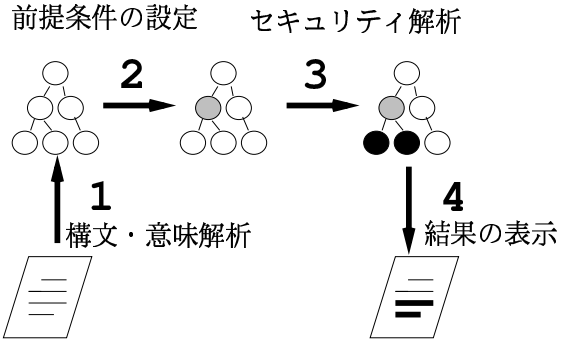

図 4: 解析の流れ

である *Osaka Slicing System*，以下，*OSS*[9] に，3. で述べたセキュリティ解析部を機能追加する形で行った．

4.1 ツールの概要

ツールの解析の流れを図 4 に示す．構文・意味解析部は，UI 部からの要求に応じて構文解析，意味解析を行なう（図 4-1）．次に，ユーザはソースファイル上でセキュリティ解析の前提条件を設定（図 4-2）し，UI 部を通じてセキュリティ解析部へ依頼する．セキュリティ解析部は，前提条件を基にセキュリティ解析を行ない（図 4-3），その結果を UI 部に渡す．UI 部は，SC の高い変数が使われる可能性のある文を強調する形で解析結果を表示する（図 4-4）．

4.2 ツールの適用事例

我々は本ツールの利用目的として，プログラムの安全性の確認を考えている．プログラムの安全性の確認とは，プログラムを静的に解析し SC の高い出力を事前に検出することで，予想外の情報漏洩を防ぐことをいう．プログラムの安全性を確認し対策を立てることで，SC の高い出力文を減らし，情報漏洩の可能性をより低くすることができる．

適用事例として，飛行機の予約システムを考える．このシステムにはクレジットカードの認証を行なうモジュールが組み込まれている．

クレジットカード番号に関する情報にのみ高い SC を与えて解析を行なった結果，システム全体で 36 個ある出力文のうち，35 個の出力文が高い SC を持つという結果が得られた（図 5）．これらの出力文中には，予約処理モジュールの出力文も含まれていた．これは，このシステムが図 6 のように認証が成功した後に予約を行なうよう実装され

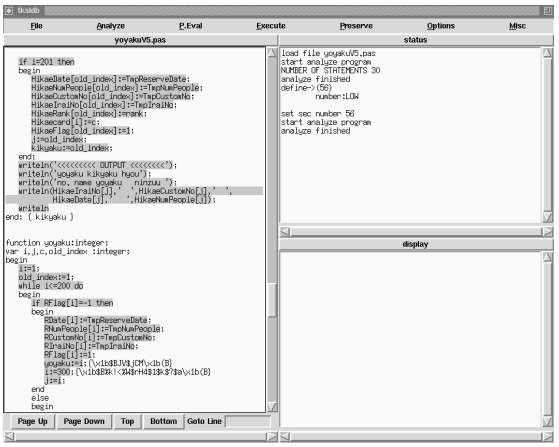

図 5: 変更前の解析結果

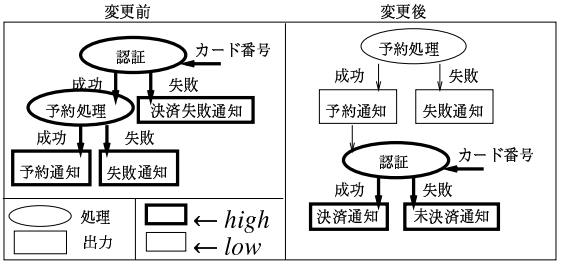

図 6: 予約システムの制御の流れ

ていたためである．このような実装では，「予約処理が行なわれる」ことが「与えられたクレジットカード番号が認証された」ことを示しており，クレジットカードの番号に関する情報が漏洩していることになる．

そこで，クレジットカード番号と予約処理モジュールの出力文に存在する情報フローを排除するため，認証前に予約処理を行なう方法に変更した（図 6）．ツールを用いて前回と同じ条件で解析を行なった結果，高い SC を持つ出力文は，クレジットカード認証に関する 13 個のみに減少させることができた（図 7）．また，予約処理モジュールの出力文が高い SC を持たないことも確認できた．

このように，実装の方法によって情報の流れは大きく変わるため，情報フロー解析による安全性の確認は重要である．

5 オブジェクト指向型言語に対する適用

現在のプログラム開発環境において，C などの手続き型言語だけでなく，C++[5] や Java[7] に代表されるオブジェクト指向言語（*Object Oriented Language*，以下，OO 言語と省略する）[6] が多く利用されるようになった．

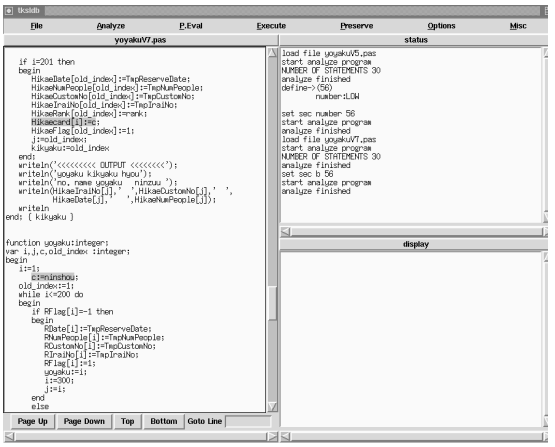

図 7: 変更後の解析結果

```
class Base {
  protected String value;
  public void method(String v) {
    value...(1) = v...(1);
  }
}
class Derived extends Base {
  public void method(String v) { value = v; }
}
class Sample {
  public static void main(String[] args) {
    Base x = new Derived();
    ...
    x.method(args[0]);        // args[0] ← high
  }
}
```

図 8: オーバーライドに関する問題

OO 言語には，手続き型言語には無い，クラス（*Class*），継承（*Inheritance*）などの新たな概念が導入され，既に提案されている手続き型言語に対するセキュリティ解析アルゴリズムをそのまま適用すると様々な問題が生じる．本節では，オブジェクト指向特有の概念から生じる問題点をいくつか挙げ，それぞれに対し Java プログラムを具体例に用いながら対策方針を述べる．

継承，オーバーライド，オーバーロード

継承によりメソッドのオーバーライド（*Override*）[¶1]が起こると，同じクラス階層にシグニチャ（*Signature*）[¶2]の等しいメソッドが複数存在することになる．図 8 では，メソッド Base::method(String v) が Derived::method(String v) によりオーバーライドされているのが分かる．このとき，文

[¶1] 派生クラスにおいて，基底クラスに存在する同一シグニチャのメソッドを再定義する [7]

[¶2] メソッド名および引数の型 [7]

```
class Class {
  protected String value;
  public void method(String v) { value = v }
  public String method_() { return(value); }
}
class Sample {
  protected String value;
  public static void main(String[] args) {
    Class x = new Class(); Class y = new Class();
    x.method(args[0]);        // args[0] ← high
    System.out.println(x.method_());
    y.method(args[1]);        // args[1] ← low
    System.out.println(y.method_()...(2));
  }
}
```

図 9: インスタンスに関する問題

x.method(args[0]) で利用されている参照変数 x は Base クラス型であるため，Base クラスおよび Derived クラスいずれのインスタンスも参照することができる．そのため，x.method() の実体が Base::method()，Derived::method() のどちらであるのか判断できず，解析結果の安全性を満たすには両者が呼び出されるとみなさねばならない．その結果，図 8 の網掛部の SC は *high* であると判定される．

そこで，既に提案されているオブジェクト指向プログラムに対するエイリアス解析手法 [10] を利用し参照変数の指すインスタンスの型を特定することで，解析対象となるオーバーライドメソッドを限定することができる．図 8 の場合，エイリアス解析により参照変数 x が Derived クラスのインスタンスを指していることが分かり，x.method() の実体が Derived::method() であると決定できる．そのため，Base::method() は解析対象から除外され，太枠部 $_{...(1)}$ の SC が実際には *low* であるとみなすことができる．

ここではオーバーライドに関してのみ述べたが，メソッドのオーバーロード（*Overload*）¶3においてもエイリアス解析による解析精度の向上が期待できる．

クラス，インスタンス

一般にクラスのインスタンスは複数存在する．インスタンス内部のセキュリティ情報をクラス単位で共有する場合，あるインスタンス i の属性 a に関するセキュリティ情報は，それよりも SC の高い，同一クラスの異なるインスタンス i' の属性 a により破棄されてしまう．図 9 では，文 x.method(args[0]) により Class::value の SC が *high* になると，参照変数 x および y が異なるインスタンスを指し，かつ args[1] の SC が *low* であるにもかかわらず，y.method_() の SC は *high* となる．

¶3名前が同じで引数の型の異なるメソッドを複数定義し，引数の型に応じてメソッドが選択される [7]

```
class A {
  private String value;
  public void method(String v) { value = nil; }
}
class B {
  private String value;
  public void method(String v) { value = v; }
}
class Sample {
  public static void main(String[] args) {
    A x...(3) = new A()...(3);
    B y...(4) = new B()...(4);
    x...(3).method(args[0]);      // args[0] ← high
    y...(4).method(args[0]);
  }
}
```

図 10: インスタンス属性に関する問題

そこで，インスタンス内部のセキュリティ情報をインスタンス単位で保持する手法が考えられる．図 9 の場合，x.value と y.value のセキュリティ情報を区別することで，太枠部 $_{...(2)}$ の SC が *low* と判定できる．

インスタンス，インスタンス属性

オブジェクト指向プログラムで利用される各値の SC を求めるにあたって，インスタンス自身の SC と そのインスタンスが持つ属性の SC との関係を定義する必要がある．ここでは，「インスタンス自身の SC は，

- インスタンス属性の SC
- インスタンスメソッドの存在 および それらに渡される実引数の SC

により決定される」という基本方針による，3 つの SC の定義規則を述べる．

(1) インスタンス属性が左辺となる代入文における右辺値の **SC** および インスタンスメソッドの実引数の **SC** の最小上界
最も実現が容易な方法である．しかし，
・引数の値を特定のルールに従って変換しそれを戻り値として出力する，かつ
・属性に影響を与えない
ようなメソッドを多く持つ，共通ルーチンとしての利用度が高いクラスのインスタンスの場合，その SC は高くなりやすい．
図 10 の網掛部は，この規則により SC が *high* と判定されたものを表している．

(2) インスタンス属性の **SC** の最小上界

(1) では，インスタンスメソッドの実引数の SC に直接影響を受けていたが，本規則を適用することでインスタンス属性に影響を与えない実引数の SC を計算対象から排除することができる．ただし各メソッドにおいて，引数とインスタンス属性間の情報フローを把握する必要がある．

図 10 において，文 x.method(args[0]) で呼ばれる A::method() メソッドは属性 A::value に影響を与えないと判断でき，参照変数 x が指すインスタンスの SC は low となる．これにより，太枠部 $_{...(3)}$ が low の SC を保持するとみなすことができる．

(3) （メソッドを介してその情報フローがインスタンス外に到達する可能性のある）インスタンス属性の **SC** の最小上界

(2) では，すべてのインスタンス属性の SC の最小上界をインスタンス自身の SC としたが，インスタンス属性の中にはその情報フローがインスタンス外部に流れないものも存在する．本規則では，そのようなインスタンス属性の SC は計算対象から排除する．ただし，各属性の情報フローがどのメソッドを介して外部に流れるかを前もって把握しておく必要がある．

図 10 において，文 y.method(args[0]) で呼ばれる B::method() メソッドは属性 B::value の値を定義する．しかし，その値に関する情報フローは決して外部に流れないと判断でき，参照変数 y が指すインスタンスの SC は low となる．これにより，太枠部 $_{...(4)}$ が low の SC を保持するとみなすことができる．

以上，OO 言語へのセキュリティ解析アルゴリズム適用における問題点をいくつか挙げ，それぞれに対する対策方針を述べた．上記の方針を適用することで精度の高い解析結果を得ることはできるが，その計算コストも考慮しなければならない．これらはトレードオフの関係にあるため，実利用における手法の比較検討が必要である．

6 まとめ

本研究では，[8] で提案されたセキュリティモデルに基づく情報フロー解析アルゴリズムの実装を行い，手法の有効性を検証した．実装においては，実利用を考慮して，解析の効率化，大域変数への対応を行った．また，オブジェクト指向言語に対してセキュリティ解析アルゴリズムを適用する際の問題点を挙げ，その対処方法について考察した．

今後の課題としては，

- オブジェクト指向言語に対するセキュリティ解析アルゴリズムの実装
- 解析アルゴリズムの違いによる，精度とコストのトレードオフ関係の検証
- 束構造のセキュリティモデルに基づく解析手法の実現

などが挙げられる．

参考文献

[1] D.E.Denning: “A Lattice Model of Secure Information Flow”, Communication of the ACM,Vol. 19, No. 5, pp. 236–243, 1976.

[2] D.E.Denning and P.J.Denning: “Certification of Programs for Secure Information Flow”, Communication of the ACM,Vol. 20, No. 7, pp. 504–413, 1977.

[3] J.Banâtre, C.Bryce and D.Le Métayer: “Compile-Time Detection of Information Flow in Sequential Programs”, Proc.3rd ESORICS, LNCS 875, pp. 55–73, 1994.

[4] G.Purnul: “Database Security”, Advances in Computers(M.Yovits Ed.),Vol. 38, pp. 1–72, 1994.

[5] B. Stroustrup: “The C++ Programming Language(Third edition)”, Addison-Wesley, 1997.

[6] G. Booch: “Object-Oriented Design with Application”, The Benjamin/Cummings Pubrishinh Company, Inc, 1991.

[7] J. Gosling,1 B. Joy, and G. Steele, 村上 雅章 [訳]: “The JAVA 言語仕様”

[8] 國信, 高田, 関, 井上 : “束構造のセキュリティモデルに基づくプログラムの情報フロー解析”, 電子情報通信学会技術研究報告,2000 年 11 月.

[9] 佐藤, 飯田, 井上: “プログラムの依存関係解析に基づくデバッグ支援システムの試作”, 情処学論, Vol. 37, No. 4, pp. 536–545, 1996.

[10] 大畑, 井上: “オブジェクト指向プログラムにおけるエイリアス解析について”, 情処学研報, 2000-SE-126, pp.57–64, 2000.