修士学位論文

題目

オブジェクト指向プログラムにおける
セキュリティ解析アルゴリズムの提案と実現

指導教官

井上克郎 教授

報告者

横森 励士

平成 13 年 2 月 14 日

大阪大学 大学院基礎工学研究科

情報数理系専攻 ソフトウェア科学分野

平成 12 年度 修士学位論文

オブジェクト指向プログラムにおける
セキュリティ解析アルゴリズムの提案と実現

横森 励士

内容梗概

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．このようなシステム上での情報漏洩を防ぐために，機密度の高いデータを扱うプログラムが情報の漏洩を引き起こさないことを保証する手法として，セキュリティ解析アルゴリズムが提案されている．セキュリティ解析アルゴリズムでは，まずプログラムを静的に解析することによりプログラム中の変数間に存在するデータ授受関係を抽出する．更にプログラムへの入力にセキュリティクラス (情報の重要度) を与えることで，プログラム中で扱われる値のセキュリティクラスの導出を行う．このアルゴリズムは，手続き型言語を対象として提案されているが，既存の研究においてはその提案のみにとどまっており，その実現については触れられていない．

そこで，本研究では，手続き型言語を対象としたセキュリティ解析アルゴリズムを実装し，適用事例を通じてその有効性を検証する．また，現在のプログラム開発環境において，手続き型言語だけでなく，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おいてはクラス，インスタンスなど特有の概念が存在し，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ことはできない．そこで更に本研究では，手続き型言語を対象としたセキュリティ解析アルゴリズムの拡張により，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を新たに提案する．また，提案した手法の一部を実際のツールとして実現し，その手法の正当性を確かめる．

主な用語

セキュリティ解析 (security analysis)
情報フロー解析 (information flow analysis)
手続き型言語 (procedural language)
オブジェクト指向型言語 (object-oriented language)

目 次

1 まえがき **4**

2 セキュリティ解析 **5**

3 手続き型言語におけるセキュリティ解析 **7**

3.1 実装の方針 7

3.2 解析の手順 7

3.2.1 大域変数の解析 9

3.2.2 手続き内解析 9

3.2.3 手続き間解析 10

3.2.4 解析コスト 10

3.2.5 解析例 14

3.3 実装 14

3.4 ツールの概要 14

3.5 ツールの適用事例 16

4 オブジェクト指向型言語におけるセキュリティ解析 **20**

4.1 実装の方針 20

4.2 解析の手順 20

4.2.1 メソッド内解析 20

4.2.2 参照変数の解析 21

4.2.3 メソッド間解析 28

4.2.4 解析コスト 28

4.2.5 解析例 29

4.3 実装 31

4.4 ツールの適用事例 32

5 オブジェクト指向型言語におけるセキュリティ解析の問題点とその対策手法 **33**

5.1 対策手法の実装 36

5.1.1 オーバーライドに関する問題に対する対策手法の実現 36

6 まとめ **39**

謝辞 **40**

参考文献 41

1　まえがき

近年ネットワークを利用した商取引が盛んになりつつあり，個人と企業間でのオンライン上での商取引が盛んになっている．オンライン上での取引では，決済の際にクレジットカードに関する情報や個人に関する情報を入力し，決済を行なっている．このような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システムやそのシステム上で動作するプログラ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．Denning らは [5, 6] 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，國信ら [10] によって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, [10] 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，[10] 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，C++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．そこで，手続き型言語を対象としたセキュリティ解析アルゴリズムを拡張して，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を提案する．提案したアルゴリズムは，オブジェクト指向言語特有の特有のクラス，インスタンス，継承などの新しい概念に対応できるように工夫している．提案した手法を Java プログラムを対象としたセキュリティ解析ツールとして実現し，対策手法の内一つをセキュリティ解析ツールに対して実現する．

以降，2. ではセキュリティ解析について述べる．3. で手続き型言語におけるセキュリティ解析アルゴリズムの実装について述べ，ツールの概要および適用事例を紹介する．

また 4. でオブジェクト指向型プログラムにおけるセキュリティ解析アルゴリズムについて述べ，ツールの概要を紹介する．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

近年ネットワークを利用した商取引が盛んになりつつあり，個人と企業間でのオンライン上での商取引が盛んになっている．オンライン上での取引では，決済の際にクレジットカードに関する情報や個人に関する情報を入力し，決済を行なっている．このような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システムやそのシステム上で動作するプログラムにおいては，不適切な情報漏洩を防ぐことは重要な課題である．

このようなシステム上での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 [12]:

> 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 [5, 6] ．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，[6] 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，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，[10] 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 [10]
- 入力値の SC と出力域のクリアランスとの矛盾を検出する [6]

の 2 つの手法があるが，本稿では前者に着目する．

3　手続き型言語におけるセキュリティ解析

我々は[10]によって提案されている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を基に，情報フロー解析アルゴリズムを再定義し，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3.3.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[10]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
  [10]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

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 “)”].

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．

3.2.1 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．

3.2.2 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，図 2 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 3 には図 1，図 2 で利用される要素を示している．

3.2.3 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 [13] 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．

3.2.4 解析コスト

この節では，このアルゴリズムにおける解析コストについて考察する．

SC モデル束構造の頂点数を S，手続き数を M とするとこのアルゴリズム下では，解析の条件となる各変数の SC が変更されるたびに解析を行なう．そのため，解析の入力となる変数 (大域変数・仮引数) の最大の数を V とすると最悪の場合

$O(S \times V)$

回単一の手続きを再解析するため，最悪の場合の総手続き解析回数は

(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}

図 1: ALGORITHM(s, imp)(1/2)

(ブロック文)(being B_1; ... 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$_{next}$:= $\emptyset$

for i := 0 **to** |s$_{actuals}$| **begin**

  cl := { c | (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;

$kill$:= $\emptyset$;

for i := 0 **to** |s$_{actuals}$| **begin**

  $kill$:= $kill$ ∪

    { ($P_{formals}$[i], c) | ($P_{formals}$[i], c)∈SCset }

end

SCset := SCset − $kill$

図 2: ALGORITHM(s, imp)(2/2)

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 3: アルゴリズムの要素

$O(M \times S \times V)$

となる．

次に単一の手続きにおける文の解析回数を考える．このとき繰り返し文の構造の最大の深さを k，繰り返し文の構造の最大の深さが k である時にその繰り返し構造で判定される変数の総数を V_a，文の数を T とすると，文の総解析回数は

$O((S \times V_a)^k \times T)$

となる．よって一回の解析の最悪の場合の総解析回数は

$O(M \times V \times (S \times V_a)^{k+1} \times T)$

となる．今回の解析アルゴリズムは変数の数が少ない小規模なプログラムに対しては有効であるが，階層化しその中で判定される変数の数が多いプログラムに対しては，解析の繰り返しの条件となる変数が多くなることで繰り返し回数が増加し，効率が落ちることが考えられる．

3.2.5 解析例

例として，図 4 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

$\mathrm{SCset}(8) := \{(\mathrm{a}, low), (\mathrm{x}, low), (\mathrm{y}, low)\}$

を定義し．先頭の文 (8 行目) から解析を行なう．図 1，2 より，

$kill := \{(\mathrm{y}, low)\}$; $gen := \{(\mathrm{y}, high)\}$

$\mathrm{SCset}(8') := \mathrm{SCset}(8) - kill \cup gen$

$:= \{(\mathrm{a}, low), (\mathrm{x}, low), (\mathrm{y}, high)\}$

を求め，これを $\mathrm{SCset}(9)$ 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．

$\mathrm{SCset}(18') := \{(\mathrm{a}, high), (\mathrm{x}, low), (\mathrm{y}, high), (\mathrm{f}, high)\}$

3.3 実装

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である *Osaka Slicing System*，以下，*OSS*[13] 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．

3.4 ツールの概要

ツールの解析の流れを図 5 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 5-1)．次に，ユーザはソースファイル上でセキュリティ解析の

```
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 4: 手続き内解析の例

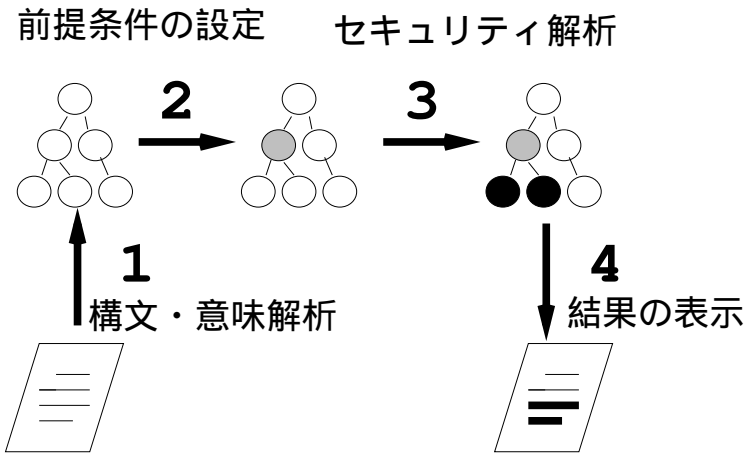

図 5: 解析の流れ

前提条件を設定 (図 5-2) 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 5-3) , 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 5-4) .

3.5 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 , チケット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 6) . 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 8 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 8) . ツールを用いて前回と同

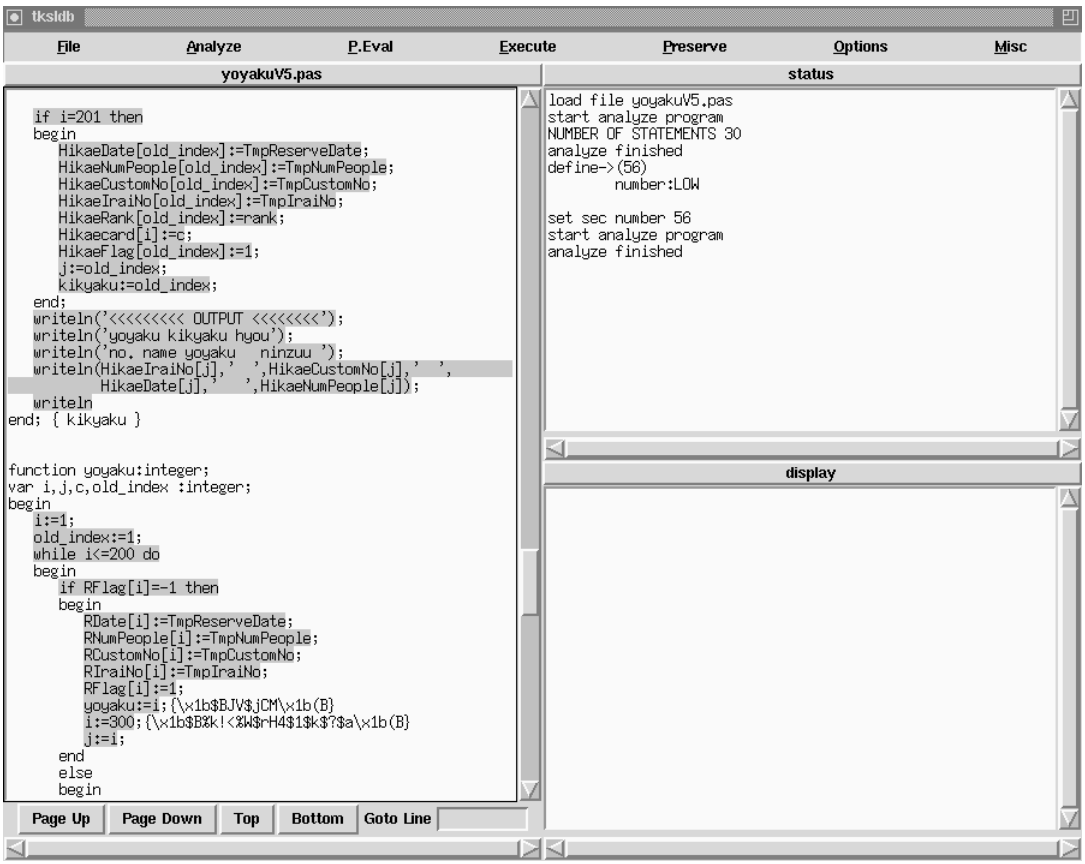
tksldb
File
Analyze
P.Eval
Execute
Preserve
Options
Misc
yoyakuV5.pas
status
display
Page Up
Page Down
Top
Bottom
Goto Line

図 6: 変更前の解析結果

図 7: 変更後の解析結果

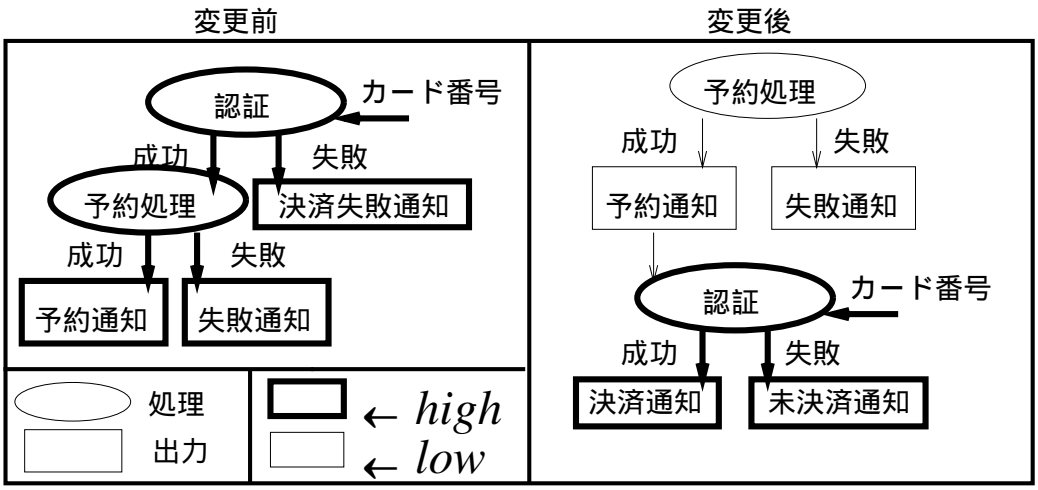

図 8: 予約システムの制御の流れ

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．

4　オブジェクト指向型言語におけるセキュリティ解析

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++[14] やJava[8] 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）[4] が多く利用されるようになった．そこで，手続き型言語を対象としたセキュリティ解析アルゴリズムを拡張することで，OO 言語に対するセキュリティ解析アルゴリズムを提案する．ツールの詳細は 4.3. で述べる．

4.1　実装の方針

対象言語は Java とする．現在のところシングルスレッドを対象とし，マルチスレッドには対応していない．また，実装においては SC として任意の束構造を想定しており，その点が前節のアルゴリズムと異なる．

Java には明示的な出力文，入力文がないため出力文は「java.io クラス中の引数に対して出力を行なうメソッド」を呼び出すメソッド呼び出し文，入力文は「java.io クラス中において値を読み込むメソッド」を呼び出すメソッド呼び出し文，とする．

4.2　解析の手順

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．

入力文に対応するメソッド:　入力文に対応するメソッドで読み込まれる値の SC

メイン関数宣言部:　仮引数の SC

また，局所変数・メンバ変数の 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，各出力文における SC を表示する．

以降，**Phase 2:** 情報フロー解析に関して，メソッド内解析，参照変数の解析，メソッド間解析に分けそれぞれ述べる．

4.2.1　メソッド内解析

メソッド内解析を行なう際には，はじめに，そのメソッドが属するクラスのメンバ変数や，仮引数などから，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．

SCset の要素は通常型の変数の場合と参照変数の場合では異なり，通常型の変数の場合には (変数, SC) の組で表現し，参照型の変数の場合には (変数，その型のインスタンスが内部に持ちうるメンバ変数からなる SCset) で表現する．

また，各変数の SC の初期値は以下の通りである．

通常型の変数と **String** 型の変数: low

参照型の変数: 参宣言されている型をインスタンス化した場合に内部に持ちうるメンバ変数からなる SCset

通常型の仮引数: 対応するメソッド呼び出し文の実引数の SC の上界

参照型の局所変数: 対応するメソッド呼び出し文の実引数が持ちうる SCset の和集合，各要素毎に上界をとる．

その後，メソッド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 9 ~ 13 に基づき，, メソッドの先頭の文を M_{start} とすると，$\textsc{Algorithm}(M_{start}, \emptyset)$ 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

$$\frac{\text{s の解析時の内部処理}}{\text{s の解析終了時の SCset および 出力文の持つ SC}}$$

の形式で記述している．図 14 には図 9 ~ 13 で利用される要素を示している．

4.2.2 参照変数の解析

参照型の変数はその変数自身の SCset の代わりにその変数が指しうるインスタンスが持ちうるメンバ変数からなる SCset を持つ．参照型の変数に対する参照が行なわれた場合はその変数が持っている SCset に対して解析を行なう．また，参照型の代入文では右辺が指す参照変数が指す SCset と左辺が指す参照変数が指す SCset を共有させる．この時指す SCset は右辺が指す参照変数が指していた SCset となる．こうすることで実際に解析上でエイリアスが起こり，プログラムの実行時にエイリアスが起こる場所での解析が正確になる．

子に自分自身の型を持つ場合などのように，同じ名前の変数が複数回出現し，無限にループする場合がある．その際は，子が指す SCset はクラス・名前・型が同じ親が持つ SCset と同一とし，SCset を親子間で共有するものとする．

(空文)

$$\frac{}{\mathrm{SCset} := \mathrm{SCset}}$$

(基本型 (またはそれに準ずる型の) 代入文)

$$\frac{\begin{array}{l} cl := \{\ \mathrm{c} \mid \mathrm{x}{\in}\mathrm{Ref(s)} \wedge (\mathrm{x},\ \mathrm{c}){\in}\mathrm{SCset}\ \} \cup imp; \\ kill := \{\ (\mathrm{x},\ \mathrm{c}) \mid \mathrm{x}{\in}\mathrm{Def(s)} \wedge (\mathrm{x},\ \mathrm{c}){\in}\mathrm{SCset}\ \}; \\ gen := \{\ (\mathrm{x},\ \sqcup_{c \in cl} c) \mid \mathrm{x}{\in}\mathrm{Def(s)}\}; \end{array}}{\mathrm{SCset} := \mathrm{SCset} - kill \cup gen}$$

(参照型の代入文)

$$\frac{\begin{array}{l} kill := \{\ (\mathrm{x},\mathrm{SCset(x)}) \mid \mathrm{x}{\in}\mathrm{Def(s)}\}; \\ gen := \{\ (\mathrm{x},\mathrm{SCset(right)})\}; \end{array}}{\mathrm{SCset} := \mathrm{SCset} - kill \cup gen}$$

(入力に対応するメソッド)

$$\frac{\begin{array}{l} kill := \{\ (\mathrm{x},\ \mathrm{c}) \mid \mathrm{x}{\in}\mathrm{Def(s)} \wedge (\mathrm{x},\ \mathrm{c}){\in}\mathrm{SCset}\ \}; \\ gen := \mathrm{SCset}_{input}(\mathrm{s}) \\ \qquad (*\ \{\ \mathrm{x} \mid \mathrm{x}{\in}\mathrm{SCset}_{input}(\mathrm{s})\ \} = \mathrm{Def(s)}\ *) \end{array}}{\mathrm{SCset} := \mathrm{SCset} - kill \cup gen}$$

(出力に対応するメソッド)

$$\frac{\begin{array}{l} \textsc{Algorithm}(\mathrm{Exp(s)},\ \sqcup_{c \in cl} c); \\ cl := \{\ \mathrm{c} \mid \mathrm{x}{\in}\mathrm{Ref(s)} \wedge (\mathrm{x},\ \mathrm{c}){\in}\mathrm{SCset}\ \} \cup imp \end{array}}{\mathrm{SC}_{output} := \sqcup_{c \in cl} c;\ \mathrm{SCset} := \mathrm{SCset}}$$

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

$$\frac{\begin{array}{l} \textsc{Algorithm}(\mathrm{B}_1,\ imp); \\ \ldots \\ \textsc{Algorithm}(\mathrm{B}_n,\ imp) \end{array}}{\mathrm{SCset} := \mathrm{SCset}}$$

ただし，途中に確実に実行される break 文 continue 文 return 文があるなら，それ以降の解析を中止する．

図 9: Algorithm(s, *imp*)(1/5)

(繰り返し文)(while E do B)

ALGORITHM(Exp(E),imp);
SCset_{pre} := SCset;
repeat
  cl := { c | x∈Ref(E)∧(x, c)∈SCset } ∪ imp;
  ALGORITHM(B, $\sqcup_{c \in cl} c$);
  ALGORITHM(Exp(E), $\sqcup_{c \in cl} c$);
  SCset_{pre} := unite(SCset, SCset_{pre})
until SCset = SCset_{pre}

SCset := SCset_{pre}

(繰り返し文)(do B while E)

SCset_{pre} := SCset;
repeat
  cl := { c | x∈Ref(E)∧(x, c)∈SCset } ∪ imp;
  ALGORITHM(B, $\sqcup_{c \in cl} c$);
  ALGORITHM(Exp(E), $\sqcup_{c \in cl} c$);
  SCset_{pre} := unite(SCset, SCset_{pre})
until SCset = SCset_{pre}

SCset := SCset_{pre}

(繰り返し文)(for(B_{init} ; E ; B_{kousin}) B)

ALGORITHM(B_{init},imp);
ALGORITHM(Exp(E),imp);
SCset_{pre} := SCset;
repeat
  cl := { c | x∈Ref(E)∧(x, c)∈SCset } ∪ imp;
  ALGORITHM(B, $\sqcup_{c \in cl} c$);
  ALGORITHM(B_{kousin}, $\sqcup_{c \in cl} c$);
  ALGORITHM(Exp(E), $\sqcup_{c \in cl} c$);
  SCset_{pre} := unite(SCset, SCset_{pre})
until SCset = SCset_{pre}

SCset := SCset_{pre}

図 10: ALGORITHM(s, imp)(2/5)

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

ALGORITHM(Exp(E), $\sqcup_{c \in cl} c$); SCset_{exp} := SCset;
cl := { c | x∈Ref(E)∧(x, c)∈SCset } ∪ imp;
SCset := SCset_{exp};
ALGORITHM(B_{then}, $\sqcup_{c \in cl} c$); SCset_{then} := SCset;
SCset := SCset_{exp};
ALGORITHM(B_{else}, $\sqcup_{c \in cl} c$); SCset_{else} := SCset;

SCset := unite(SCset_{then}, SCset_{else})

(分岐文)(switch E case B1,. . .,case Bn)

ALGORITHM(Exp(E), $\sqcup_{c \in cl} c$); SCset_{exp} := SCset;
cl := { c | x∈Ref(E)∧(x, c)∈SCset } ∪ imp;
SCset := SCset_{exp};
ALGORITHM(B_1, $\sqcup_{c \in cl} c$);
: (*)
ALGORITHM(B_n, $\sqcup_{c \in cl} c$); SCset_{B_1} := SCset;
SCset := SCset_{exp};
ALGORITHM(B_2, $\sqcup_{c \in cl} c$);
: (*)
ALGORITHM(B_n, $\sqcup_{c \in cl} c$); SCset_{B_2} := SCset;
:
:
:
SCset := SCset_{exp};
ALGORITHM(B_n, $\sqcup_{c \in cl} c$); SCset_{B_n} := SCset;

SCset := unite(SCset_{B_1},. . ., SCset_{B_n});

(*) ただし，途中に確実に実行される break 文 continue 文 return 文があるなら，それ以降の解析を中止する．

図 11: ALGORITHM(s, imp)(3/5)

(ブロック文)(being B_1; ... B_n; end)

ALGORITHM(B$_1$, imp);

...

ALGORITHM(B$_n$, imp)

SCset := SCset

(synchronized 文)(synchronized E B)

ALGORITHM(Exp(E), $\sqcup_{c \in cl} c$); SCset$_{exp}$:= SCset;

cl := { c | x∈Ref(E)∧(x, c)∈SCset } ∪ imp;

SCset := SCset$_{exp}$;

ALGORITHM(B, $\sqcup_{c \in cl} c$); SCset$_{out}$:= SCset;

SCset := unite(SCset$_{out}$, SCset$_{exp}$)

(try 文)(try B_t catch $C_1 \ldots C_n$ finallyB_f

ALGORITHM(Exp(B_t), imp); SCset$_{B_t}$:= SCset;

cl := { c | x∈Ref(B_t)∧(x, c)∈SCset } ∪ imp;

SCset := SCset$_{B_t}$;

ALGORITHM(C$_1$, $\sqcup_{c \in cl} c$); SCset$_{B_1}$:= SCset;

:

SCset := SCset$_{B_t}$;

ALGORITHM(C$_n$, $\sqcup_{c \in cl} c$); SCset$_{B_n}$:= SCset;

SCset(C) :=SCset := unite(SCset$_{B_1}$,..., SCset$_{B_n}$);

ALGORITHM(B$_f$, $\sqcup_{c \in cl} c$); SCset$_{B_f}$:= SCset;

SCset := unite(SCset$_{exp}$,SCset$_{B_f}$);

(throw 文)

ALGORITHM(Exp(E), $\sqcup_{c \in cl} c$); SCset$_{exp}$:= SCset;

cl := { c | x∈Ref(E)∧(x, c)∈SCset } ∪ imp;

SCset := SCset$_{exp}$

図 12: ALGORITHM(s, imp)(4/5)

(インスタンス生成式)

インスタンス生成に利用するコンストラクタを M とする．

$\mathrm{SCset}_{next} := \emptyset$
for i := 0 **to** $|\mathrm{s}_{actuals}|$ **begin**
  $cl := \{\ \mathrm{c} \mid (\mathrm{s}_{actuals}[\mathrm{i}], \mathrm{c}) \in \mathrm{SCset}\ \}$;
  $\mathrm{SCset}_{next} := \mathrm{SCset}_{next} \cup \{\ (M_{formals}[\mathrm{i}], cl)\ \}$;
end;
foreach x$\in M_{member}(M)$ **begin**
  $\mathrm{SCset}_{next} := \mathrm{SCset}_{next} \cup$
    $\{\ (\mathrm{x}, \mathrm{c}) \mid (\mathrm{x}, \mathrm{c}) \in \mathrm{SCset}\ \}$
end;
$\mathrm{SCset} := \mathrm{SCset}_{next}$;
コンストラクタ M 内の解析;
$kill := \emptyset$;
for i := 0 **to** $|\mathrm{s}_{actuals}|$ **begin**
  $kill := kill \cup$
    $\{\ (M_{formals}[\mathrm{i}], \mathrm{c}) \mid (M_{formals}[\mathrm{i}], \mathrm{c}) \in \mathrm{SCset}\ \}$
end

$\mathrm{SCset} := \mathrm{SCset} - kill \cup Def(S)$

(メソッド呼び出し文)

呼び出されうるメソッドの集合を MS とする．

$\mathrm{SCset}_{after} := \emptyset$
$\mathrm{SCset}_{begin} := \mathrm{SCset}$
foreach (M$\in MS$) **begin**
$\mathrm{SCset} := \mathrm{SCset}_{begin}$
$\mathrm{SCset}_{next} := \emptyset$
for i := 0 **to** $|\mathrm{s}_{actuals}|$ **begin**
  $cl := \{\ \mathrm{c} \mid (\mathrm{s}_{actuals}[\mathrm{i}], \mathrm{c}) \in \mathrm{SCset}\ \}$;
  $\mathrm{SCset}_{next} := \mathrm{SCset}_{next} \cup \{\ (M_{formals}[\mathrm{i}], cl)\ \}$;
end;
foreach x$\in M_{member}(M)$ **begin**
  $\mathrm{SCset}_{next} := \mathrm{SCset}_{next} \cup$
    $\{\ (\mathrm{x}, \mathrm{c}) \mid (\mathrm{x}, \mathrm{c}) \in \mathrm{SCset}\ \}$
end;
$\mathrm{SCset} := \mathrm{SCset}_{next}$;
メソッド M 内の解析;
$kill := \emptyset$;
for i := 0 **to** $|\mathrm{s}_{actuals}|$ **begin**
  $kill := kill \cup$
    $\{\ (M_{formals}[\mathrm{i}], \mathrm{c}) \mid (M_{formals}[\mathrm{i}], \mathrm{c}) \in \mathrm{SCset}\ \}$
end
SCsetafter := SCsetafter $\cup$ (SCset$- kill$)
end

SCset := SCsetafter

図 13: Algorithm(s, imp)(5/5)

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

Exp(s): 文 s の内部に存在する式

SCset(right): 式の右辺が指す SCset

$M_{actuals}$: メソッド呼び出し文 (またはコンストラクタ) M の実引数の集合

$M_{formals}$: メソッド呼び出し文 (またはコンストラクタ) M の仮引数の集合

M_{member}: メソッド呼び出し文 (またはコンストラクタ) M によって呼び出されるメソッドが属するクラスが持つメンバ変数

M_{out}: メソッド呼び出し文 (またはコンストラクタ) M 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$_{input}$**(s):** 入力文に相当するメソッド s において設定される，(変数, SC) を要素とする集合

SC$_{output}$**(s):** 出力文に相当するメソッド 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．基本型の変数が持つ SC は，A，B においてその変数の SC の最小上界とする．参照型の変数が持つ SCset は，A，B においてその変数がもつ SCset を，$SCset_A$，$SCset_B$ とすると，unite($SCset_A$, $SCset_B$) とする．

図 14: アルゴリズムの要素

4.2.3 メソッド間解析

オブジェクト指向型プログラムは複数のクラスから構成されており，各クラスに対して複数のメソッドが定義されているため，メソッド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，あるメソッド M の解析結果が，M の呼び出し元メソッド M' の解析結果に (引数や属するクラスのメンバ変数を介して) 影響を与える可能性があり，すべてのメソッドの解析結果が安定するまで，メソッド呼び出し経路上に存在するメソッドを繰り返し解析する必要がある．そのため，メソッド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 $\mathcal{C}$，$\mathcal{S}$ はすべてのメソッドに 1 つずつ存在する．解析においては，インスタンス生成の際に呼ばれるコンストラクタもメソッドと同様に扱う．オブジェクト指向言語の場合，クラスのオーバーライドにより，一つのメソッド呼びだし文から複数のメソッドが呼ばれうる場合がある．その際には，呼び出されうるメソッドそれぞれに対して SCset $\mathcal{C}$，SCset $\mathcal{S}$，解析リストの更新処理を行なう．

解析リスト:
: メソッド呼び出し経路に基づく，メソッド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ほぼ手続き型言語の場合と同じである．

メソッド処理開始時点での **SCset** $\mathcal{C}$:
: あるメソッド呼び出し文 s があったとき，対応するメソッド M を解析する際に，M 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 M の解析を行う．一方，$\mathcal{C}$ と等価であれば M の解析は行わない．

メソッド処理終了時点での **SCset** $\mathcal{S}$:
: メソッド M の解析後，M 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，M を呼び出すすべてのメソッド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．

static 変数用の **staticSCset** :
: プログラム中で static 変数が定義がされている場合は，staticSCset を用意し，static 変数に対する参照が行なわれる場合は，staticSCset からも検索を行ない，static 変数に対する定義が行なわれる場合は，staticSCset にも定義する．

4.2.4 解析コスト

この節では，このアルゴリズムにおける解析コストについて考察する．

SC モデル束構造の頂点数を S，メソッドの数を M とするとこのアルゴリズム下では，解析の条件となる各変数の SC が変更されるたびに解析を行なう．そのため，解析の入力となる変数（メンバ変数・仮引数）の最大の数を V とすると最悪の場合

$O(S \times V)$ 回単一のメソッドを再解析するため，最悪の場合の総メソッド解析回数は

$O(M \times S \times V)$

となる．

次に単一のメソッドにおける文の解析回数を考える．このとき繰り返し文の構造の最大の深さを k，繰り返し文の構造の最大の深さが k である時にその繰り返し構造で判定される変数の総数を V_a，文の数を T とすると，文の総解析回数は

$O((S \times V_a)^k \times T)$

となる．また switch 文を構造の最下層に持つ場合，T 個の文の解析時に最悪 $O(T^2)$ の解析コストがかかる．よって一回の解析の最悪の場合の総解析回数は

$O(M \times V \times (S \times V_a)^{k+1} \times T^2)$

となる．今回の解析アルゴリズムは変数の数が少ない小規模なプログラムに対しては有効であるが，階層化しその中で判定される変数の数が多いプログラムに対しては，解析の繰り返しの条件となる変数が多くなることで繰り返し回数が増加し，効率が落ちることが考えられる．

4.2.5 解析例

例として，図 15 の関数 test に対する解析を考える．まず，仮引数，メソッドが属するクラスがとりうるメンバ変数から，

SCset(10) := {(x, low), (y, low), (s,{ (x, low), (y, $high$)})...}

を定義し，先頭の文から実行順にしたがい解析を行なう．例えば，23 行目の場合，9 より

SCset(23) := {(a, $high$),(strg,$high$) (x, low), (y, low), (s,{ (x, low), (y, $high$)})...}

$kill$:= {(y, low)};

gen := {(y, $high$)}

SCset(23') := SCset(23) − $kill$ ∪ gen

:= {(a, $high$),(strg,$high$), (x, low), (y, $high$), (s,{ (x, low), (y, $high$)})...}

を求め，これを SCset(24) として次の文（24 行目）の解析を行なう．以下同様に最後の文（28 行目）まで解析していくと，次の結果が得られる．

SCset(28) := {(a, $high$),(strg,$high$), (x, low), (y, $high$), (s,{ (x, low), (y, $high$)})...}

```
1:  class sub {
2:    int x;
3:    int y;
4:    ...
5:  }
6:  ...
7:  class sample {
8:    int x;
9:    int y;
10:   public void test(sub s) {
11:     int a = 0;
12:     String strg;
13:     InputStreamReader KBD=new InputStreamReader(System.in);
14:     BufferedReader number=new BufferedReader(KBD);
15:     try {
16:       strg = number.readLine(); ← high
17:     }
18:     catch (IOException e) {
19:       System.out.println(e.getMessage());
20:     }
21:     a =Integer.parseInt(strg);
22:     x = s.x;
23:     y = s.y;
24:     a = x+y+a;
25:     system.out.println(x);
26:     system.out.println(y);
27:     system.out.println(a);
28:   }
29:   ...
30: }
```

図 15: メソッド内解析の例

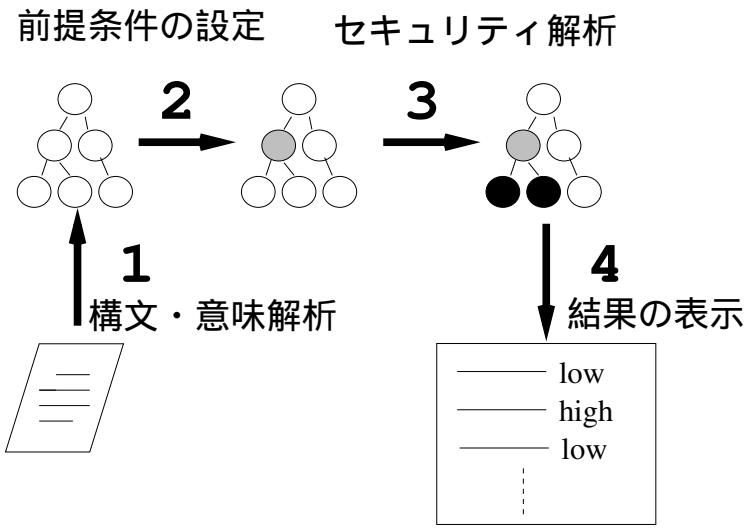

図 16: 解析の流れ

4.3 実装

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する．[11] において紹介されている Java の構文解析・意味解析ライブラリを利用して作成した．現在のところ，コード量は C ++ で 5500 行ほどとなっている．現在，CUI の形で一部について完成しており，簡単なプログラムについては解析可能となっている．

ツールの概要

ツールの解析の流れを図 16 に示す．

セキュリティ解析部

セキュリティ解析部は C++ で記述されており，[11] において紹介されている構文解析・意味解析ライブラリである libantlr・libjavamm を利用している．

構文・意味解析部は，コマンドライン上で引数として渡されたファイルに対して構文解析，意味解析を行なう (図 16-1)．次に，ユーザはその際に抽出された入出力文に関する情報を元にセキュリティ解析の前提条件を設定 (図 16-2) し，セキュリティ解析を行なう．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 16-3)，出力文に関する結果を出力する (図 16-4)．

4.4 ツールの適用事例

適用事例として，手続き型言語の場合と同じくチケットの予約システムに対して適用した場合を考える．このシステムにも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．

手続き型プログラムの場合と同様に，図 8 で示される処理の流れに基づいたチケット予約システムをそれぞれ一つずつ実装し，両者を比較した．解析では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．

「認証が成功した後に予約を行なうよう実装した」プログラムは，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，予約処理モジュールの出力文も含まれていた．一方で，認証前に予約処理を行なう方法の場合，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 個のみという結果となった．

実装の方法によって情報の流れは大きく変わり，高い SC を持つ出力文の数が増減し，保護すべき対象が変わることも考えられるため，情報フロー解析による安全性の確認は重要であると思われる．

```
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 17: オーバーライドに関する問題

5 オブジェクト指向型言語におけるセキュリティ解析の問題点とその対策手法

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++[14] やJAVA[8] 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）[4] が多く利用されるようになった．

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 17 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 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．

[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 [8]
[2] メソッド名および引数の型 [8]

```
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 18: インスタンスに関する問題

その結果，図 17 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 [11] 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 17 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 18 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．

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 18 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．

[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 [8]

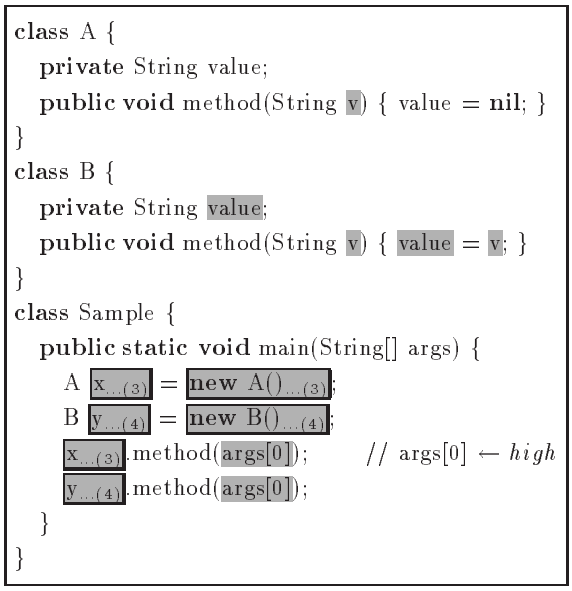

図 19: インスタンス属性に関する問題

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 19 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 19 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 19 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．

5.1 対策手法の実装

今節では，前節で述べた問題点に対する対策手法として提案されている手法のうち，一つめの

オーバーライドに関する問題

に対する対策手法を実現するアルゴリズムを説明する．

5.1.1 オーバーライドに関する問題に対する対策手法の実現

4 で紹介したアルゴリズムを次のように変更する．

- SCset の要素は通常型の変数の場合には (変数, SC) の組で表現し，参照型の変数の場合には (変数，その変数のインスタンスが持つ型, その型のメンバ変数からなる SCset) で表現する．つまり，一つの変数が型を複数持ちうる場合は複数の SCset の要素として存在する．

- **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．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．基本型の変数が持つ SC は，A，B においてその変数の SC の最小上界とする．参照型の変数が持つ SCset は，A，B においてその (変数・インスタンスが持つ型) がもつ SCset を，$SCset_A$，$SCset_B$ とすると，unite($SCset_A$, $SCset_B$) とする．

- 参照型の変数に対する参照が行なわれた場合はその変数が持っている SCset に対して解析を行なう．従来は参照変数が持っている SCset は一つだけであったが，参照変数が持っている SCset が複数になる場合がある．その時の SCset を，SCset_1,. . ., SCset_n とすると解析対象となる SCset は

  $\mathrm{SCset} = \mathrm{unite}(\mathrm{SCset}_1, \ldots, \mathrm{SCset}_n)$

  となる．

- メソッド呼び出し文では，参照変数の指すインスタンスの型を特定を行なう．つまり，変数が指すインスタンスが持ちうる型を SCset を検索し列挙する．

解析の例

例として，図 17 のクラス Sample のメソッド main に対する解析を考える .(ここでは，変数 args は通常型と同様に扱う .)

まず，仮引数，メソッドが属するクラスがとりうるメンバ変数から，

SCset := {(args, $high$)}

を定義し，先頭の文から実行順にしたがい解析を行なう．

Base x = **new** Derived(); を解析後，SCset は

SCset := {(args, $high$), (x, Derive, {(value, low)})}

となる．

その後，x.method(args[0]); を解析する際は，まず x を SCset から検索する．検索の結果，(x, derive, {(value, low)}) を検出する．次に Derive から呼び出されうる method を検索し，Derive.method() に対して情報を流す．こうすることで，Base.method() に対してメ

ソッド解析情報を流さなくなるため，実際に呼び出されない実行経路を解析から省くことが出来る．

6 まとめ

本研究では, [10] で提案されている手法を実装し，適用事例を通じて有効性を検証した．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，[10] のアルゴリズムに変更を加えた．また，手続き型言語を対象としたセキュリティ解析アルゴリズムの拡張による，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を提案した，本手法では，各内部で利用される変数をもとに SCset を作成し，解析を行なう．参照変数に対しては，そのクラスが持ちうるメンバ変数に関する SCset を持たせ，参照変数間の代入において，SCset を共有させることでエイリアスが生じた場合にも正しく解析を行なうことが出来る．また，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いることから，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た．

セキュリティ解析アルゴリズムは情報フロー解析の応用例として提案され，プログラムの安全性の確認に有効であると思われる．

今後の課題としては，

- ツールの評価
- オブジェクト指向言語に対応する際の問題に対する対処方法の実現と有効性の評価
- 例外処理への対応

などが挙げられる．

謝辞

本論文の作成において，常に適切な御指導および御助言を頂きました 大阪大学 大学院基礎工学研究科 情報数理系専攻 ソフトウェア科学分野 井上 克郎 教授に深く感謝致します．

本論文の作成において，常に適切な御指導および御助言を頂きました 奈良先端科学技術大学院大学 情報科学研究科 関 浩之 教授に深く感謝致します．

本論文の作成において，常に適切な御指導および御助言を頂きました 大阪大学 大学院基礎工学研究科 情報数理系専攻 ソフトウェア科学分野 楠本 真二 助教授に深く感謝致します．

本論文の作成において，常に適切な御助言を頂きました 奈良先端科学技術大学院大学 情報科学研究科 高田 喜朗 助手に深く感謝致します．

本論文の作成において，常に適切な御指導および御助言を頂きました 大阪大学 大学院基礎工学研究科 情報数理系専攻 ソフトウェア科学分野 松下 誠 助手に深く感謝致します．

本論文の作成およびツールの作成において，常に適切な御助言を頂きました 大阪大学 大学院基礎工学研究科 情報数理系専攻 ソフトウェア科学分野 大畑 文明 君に感謝致します．

最後に，その他の面で様々な御指導，御助言を頂いた 大阪大学 大学院基礎工学研究科 情報数理系専攻 ソフトウェア科学分野 井上研究室の皆様に深く感謝致します．

参考文献

[1] A. V. Aho, S. Sethi and J. D. Ullman, “Compilers : Principles, Techniques, and Tools”, Addison-Wesley, 1986.

[2] D. C. Atkison and W. G. Griswold, “The Design of Whole-Program Analysis Tools”, In Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, pp. 16–27, 1996.

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

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

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

[6] 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.

[7] R. Ghiya, and L. J. Hendren, “Connection Analysis: A practical interprocedural heap analysis for C”, International Journal of Parallel Programming, 24(6), pp. 547–578, 1996.

[8] J. Gosling, B. Joy, and G. Steele, 村上 雅章 [訳], “The JAVA 言語仕様”, Addison-Wesley, 2001.

[9] 片山 卓也, 土居 範久, 鳥居 宏次 [監訳], “ソフトウェア工学大事典”, 朝倉書店, 1997.

[10] 國信 茂太, 高田 喜朗, 関 浩之, 井上 克郎, “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, SS2000-30, pp. 25–32, 2000.

[11] 大畑 文明, 井上 克郎, “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”, 情報処理学会研究報告, 2000-SE-126, pp. 57–64, 2000.

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

[13] 佐藤 慎一, 飯田 元, 井上 克郎, “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”, 情報処理学会論文誌, Vol. 37, No. 4, pp. 536–545, 1996.

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

[15] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo, “Flow Insensitive C++ Pointers and Polymorphism Analysis and its Application to Slicing”, In Proceedings of the 19th International Conference on Software Engineering, pp. 433–443, 1997.

[16] M. Weiser, “Program Slicing”, In Proceedings of the 5th International Conference on Software Engineering, pp. 439–449, 1981.

[17] 横森 励士, 大畑 文明, 井上 克郎, “局所的集約によるプログラム依存グラフの効率的な構築法”, 情報処理学会第 60 回 (平成 12 年度前期) 全国大会, pp. 1-275–1-276, 2000.

[18] 横森 励士, 大畑 文明, 高田 喜朗, 関 浩之, 井上 克郎, “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”, 電子情報通信学会技術研究報告, SS2000-29, pp. 17–24, 2000.

[19] S. Zhang, B. G. Ryder, and W. A. Landi, “Experiments with Combined Analysis for Pointer Aliasing”, In Proceedings of PASTE’98, pp. 11–18, 1998.

[20] `http://www.ANTLR.org/`, “ANTLR Website.”