特別研究報告

題目

プログラム依存グラフを利用した情報漏洩解析手法の提案と実装
- 一般的な束構造を持つセキュリティクラスを対象として -

指導教官
井上 克郎 教授

報告者
西 秀雄

平成 14 年 2 月 19 日

大阪大学 基礎工学部 情報科学科

平成 13 年度 特別研究報告

プログラム依存グラフを利用した情報漏洩解析手法の提案と実装
- 一般的な束構造を持つセキュリティクラスを対象として -

西 秀雄

内容梗概

プログラムスライスは, あるデータに着目したときの, そのデータに関係する文の集合である. プログラムスライスの計算を行う高速なアルゴリズムとして, プログラム依存グラフを利用した手法が提案，実現されている．

一方で，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確認するための手法として，プログラムの入力値に設定された機密度から出力値における機密度を求める情報漏洩解析が提案，実現されている．既存の情報漏洩解析手法として実現されているのは，データフロー方程式における繰り返し計算に基づく手法である．しかし，この手法は各文毎の繰り返し計算を行う必要があるため，時間的コストがかかり，また，対応している束構造が 2 値のみのものであるなどの問題点がある．

そこで，本研究ではプログラムスライスにおける PDG 探索アルゴリズムを用いた情報漏洩解析手法を提案する．この手法では，まず PDG を構築し，構築した PDG 上で探索を行うことで解析を行う．この手法を用いることで時間的なコストを抑えることができる．さらに，本研究では一般的な束構造を持つセキュリティクラスに対する情報漏洩解析の実現方法を提案する．また，実際にシステムを試作し，適用事例を通じてその有効性を検証する．

主な用語

プログラムスライス (program slice)
プログラム依存グラフ (program dependence graph)
セキュリティクラス (security class)
情報フロー (information flow)
情報漏洩解析 (information leak analysis)

目 次

1　まえがき

プログラムスライス（*Program Slice*）とはプログラム中のある文におけるある変数の値に影響を与える全ての文，もしくはある変数の定義が影響を与える全ての文の集合である．プログラムスライスは Weiser[1] により提案され，プログラムのデバッグ支援，テスト，保守，プログラム合成などに利用されている．特に現在では，プログラム依存グラフ (*Program Dependence Graph,PDG*) と呼ばれる，プログラム文間の依存関係を表す有向グラフを用いた，より高速で正確なスライスの計算手法が存在する．

一方で，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ける不適切な情報漏洩防止を目的として，Denning らはプログラムを静的に解析し，プログラム内のデータ授受関係を表す**情報フロー**（*Information Flow*）に基づき，入力値が与えられたセキュリティに関するポリシーを保っているかどうかを検証することで，不適切な情報漏洩を検出する手法を提案した [3].

また，より現実的な手法として，プログラムに入力される値の機密度からプログラム中の各出力における機密度を調査する手法が，**情報漏洩解析**（*Information Leak Analysis*）として國信ら [16] によって提案され，我々の研究グループでは提案されている手法を [17] によって実現した．

[17] では，この両者の手法の解析の性質が極めて類似していることに着目し，プログラムスライスの応用例として [16] で提案されている手法を実現している．しかしながら，その手法ではデータフロー方程式 [8] における繰り返し計算に基づいて情報漏洩解析を行っており，時間的なコストがかかる，プログラム言語ごとにアルゴリズムを定める必要がある，などの問題点がある．また，対応しているセキュリティクラスが 2 値のみであるといった問題点もある．

そこで，本研究ではフォワードスライスを求めるアルゴリズムを利用し，PDG を探索することにより情報漏洩解析を行う手法を提案する．この手法を用いる場合，一度 PDG を構築してしまえば，プログラムスライスの場合と同様に，PDG を探索することで解析を行うことができる．そのため，解析の時間的コストを抑えることが可能となる．また，一般的にスライスで用いられる手法を利用することができ，他のプログラム言語への移植も容易となる．

さらに，これまでの情報漏洩解析では実現されていなかった，一般的な束構造をもつセキュリティクラスに基づく情報漏洩解析手法を提案する．一般的な束構造を実現する方法として，束構造上での和演算を行列として表現し，あらかじめ保存する．演算の際にはその行列を逐次参照することで，束構造上での和演算を行うことができる．

また，提案した実現手法に基づいて実際にシステムのプロトタイプを作成し有効性を検証する．評価では，前手法との実行時間の評価，例題に対する適用を行う．

以降，2 では PDG を用いたプログラムスライス計算手法について述べ，3 では既存の情報漏洩解析の手法について，4 では新たな解析手法について述べる．5 ではシステムの実現を行い，6 で試作したシステムの検証を行う．

2　プログラムスライス

プログラムスライシング(*Program Slicing*) 技術とは，プログラム P 中のある文 s におけるある変数 v に対して,v の値に影響を与える全ての文，もしくは，s における変数 w の定義に対して w が影響を与える全ての文の集合 Q を求めるための技術である．この文の集合をプログラムスライスまたは単に**スライス**(*Slice*) と呼ぶ．特に前者を**バックワードスライス**(*Backward Slice*)，後者を**フォワードスライス**(*Forward Slice*) と呼ぶ．プログラムスライシング技術は，Mark Weiser[1] によって提案され，当初はプログラムのデバッグを支援するために使われていたが，現在では，デバッグだけでなくテストや保守，プログラム合成などにも利用されている [7, 10, 11, 12].

2.1　スライス計算手順

スライスの計算には次のような手順をたどる．

Phase 1: 依存関係解析 (データ依存関係，制御依存関係）

Phase 2: プログラム依存グラフ (*Program Dependence Graph,PDG*) の構築

Phase 3: PDG を探索しスライスを抽出

2.1.1　Phase 1:依存関係解析

Phase1 では，まずソースコードを入力とし，プログラム中の文間の依存関係解析を行う．依存関係には以下の制御依存関係とデータ依存関係の 2 種類がある．

- 制御依存関係
  今，ソースプログラム p 中の文 s_1，s_2 について考える．以下の条件を全て満たしているとき，文 s_1 から文 s_2 への**制御依存** *(Control Dependence，CD)* があるという．
  - 文 s_1 が条件文である．
  - 文 s_2 が実行されるかどうかは，文 s_1 の結果に依存する．
- データ依存関係
  今，ソースプログラム p 中の文 s_1，s_2 について考える．以下の 3 つの条件を全て満たすとき，文 s_1 から文 s_2 へ変数 v に関して**データ依存** *(Data Dependence，DD)* があるという．
  - 文 s_1 で変数 v を定義している．

– 文 s_2 で変数 v を参照している．
– 文 s_1 から文 s_2 への実行可能なパスで，そのパスにおいて文 s_1 から文 s_2 間に変数 v を再定義している文が存在しない，というものが存在する．

スライスのうち，依存関係解析を全ての可能な入力データに関して行なったスライスを**静的スライス**[1](2.2 節参照）という．

2.1.2 Phase 2:プログラム依存グラフの構築

Phase2 では，Phase1 での依存関係解析で得られた依存関係情報を元に，**プログラム依存グラフ**(Program Dependence Graph．以降，PDG と呼ぶ) を構築する．PDG とは，プログラム内の文間の依存関係を有向グラフで表現したものであり，その頂点は，プログラムに含まれる条件判定部文，代入文，入出力文，手続き呼びだし文を表し, その有向辺は２つの頂点間の制御依存およびデータ依存関係を表す (それぞれ制御依存辺，データ依存辺と呼ぶ)．また，手続き間にわたるデータ依存関係を表現するための特殊頂点及び特殊辺も存在する [2].

2.1.3 Phase 3:スライス抽出

Phase3 では，Phase2 までに構築された PDG を用いてスライスを抽出する．スライスを計算抽出するには，まず『どの文のどの変数に対してスライスを計算するか』を指定する．これをスライシング基準と呼ぶ．スライシング基準は，文と変数の組から成る．このスライス基準から，PDG を探索し，到達可能な頂点の集合をスライスとして検出する．

2.2 静的スライス

2.2.1 静的スライスの計算

静的スライスの計算方法として [15] のアルゴリズムを用いる．このアルゴリズムは [5, 14] を参考にしてプログラム中の文の依存関係を調べ，それを元に**静的プログラム依存グラフ**(*PDG*) を作成し，その辺をたどることにより静的スライスを計算する．PDG は有向グラフであるので，そのたどり方にも二種類あり，有向辺を順方向にたどっていくことにより計算した静的スライスは Forward Slice，逆方向にたどっていくことにより計算した静的スライスは Backward Slice となる．

- **諸定義**

PDG はプログラム内の文の依存関係を表すグラフである．PDG の頂点はプログラム中の各文及び if 文や while 文の条件判定部分を表し，辺は変数の影響を伝える**データ依存関係.** 及び条件文や繰り返し文の制御の影響を伝える**制御依存関係**を表す.

データ依存関係は，各頂点の到達定義集合 (Reaching Definitions，略して RD) を求めることによって得られる．PDG 上でのある頂点 t の RD とは，変数 v と頂点 s との組 $\langle v, s\rangle$ の集合である．これは，

1. プログラム中の文 s で変数 v を定義している．
2. プログラム中の二つの文 s と t の間に v を必ず定義するような文がない．

ことを示している．t の RD に $\langle v, s\rangle$ が含まれ，かつ t が v を参照する時，s から t へのデータ依存関係 (DD(s, v, t)) があるという．

また，ある条件判定部分 s の結果により文 t の実行の有無が決定される時，s と t との間に制御依存関係 (CD(s, t)) が存在するものとする．すなわち制御依存関係は if 文や while 文の条件判定部分からそれらの内部ブロックに属する文への影響であり，これはプログラムを解析すれば容易に求められる．

一般にプログラムには複数の手続きが定義されている．各手続き間には引数や大域変数を通じてデータ依存関係が生じる． これらのデータ依存関係を表すために，PDG にプログラム中の文とは直接対応しない頂点 (特殊頂点と呼ぶ) を用意する．

PDG の作成は，プログラムを解析し，プログラムの各文を PDG の頂点に切りわけ，プログラム中の各文における RD を求め，それをもとにして PDG の各辺を生成することによって行なわれる.[14]

上記の方法で生成された PDG を G，G に含まれる全ての辺の集合を E とすると，プログラム内の文 S の変数 v に関するスライスを表す頂点の集合 V は以下のようにして計算される．

1. $V \leftarrow \{n | n$ は S に対応する頂点. $\}$
2. $N \leftarrow \{n | \langle v, n\rangle \in RD_{in}(S)\}$
3. $N \neq \phi$の間以下の動作を繰り返す．
   - $l \in N$ を一つ選ぶ．
   - $N \leftarrow N - \{l\}$
   - $V \leftarrow V \cup \{l\}$
   - $N \leftarrow N \cup \{k | (k \notin V) \wedge (k \xrightarrow{\star} l \in E \vee k \dashrightarrow l) \wedge (k$ が $g\text{–}in$ でない$) \wedge (k$ が $para\text{–}in$ 頂点でない$)\}$(ただし $k \xrightarrow{\star} l$ は任意の変数の k から l へのデータ依存関係辺を示す.)
   - k が $para\text{–}in$ 頂点でも $para\text{–}in$ 頂点でもないなら
     $N \leftarrow N \cup \{k | (k \notin V) \wedge (k \xrightarrow{\star} l \in E \vee k \dashrightarrow l)\}$(ただし $k \xrightarrow{\star} l$ は任意の変数の k から l へのデータ依存関係辺を示す．)

```
 1:  #include <stdio.h>
 2:  #define SIZE 5
 3:
 4:  int cube(int x) {
 5:      return x*x*x;
 6:  }
 7:
 8:  void main(void)
 9:  {
10:      int a[SIZE];
11:      int b[SIZE];
12:      int c, d, i;
13:
14:      a[0] = 0;
15:      a[1] = -1;
16:      a[2] = 2;
17:      a[3] = -3;
18:      a[4] = 4;
19:
20:      for (i=0;i<SIZE;i++) {
21:          b[i] = a[i];
22:      }
23:
24:      printf("Input: ");
25:      scanf("%d", &c);
26:
27:      if (c >= SIZE) {
28:          c = c % SIZE;
29:      }
30:
31:      d = cube(b[c]);
32:
33:      if (d < 0) {
34:          d = -1 * d;
35:      }
36:
37:      printf("%d\n", d);
38:  }
```

図 1: サンプルソースコード

2.2.2 静的スライスの適用例

図 1 に C 言語で書かれたサンプルプログラムを示す．2 はサンプルプログラムの静的スライスを計算するときに構築される PDG である．また，サンプルプログラムにおいて，スラ

イス基準 $(37, c)$ として計算した静的スライスを図 3 に示す.

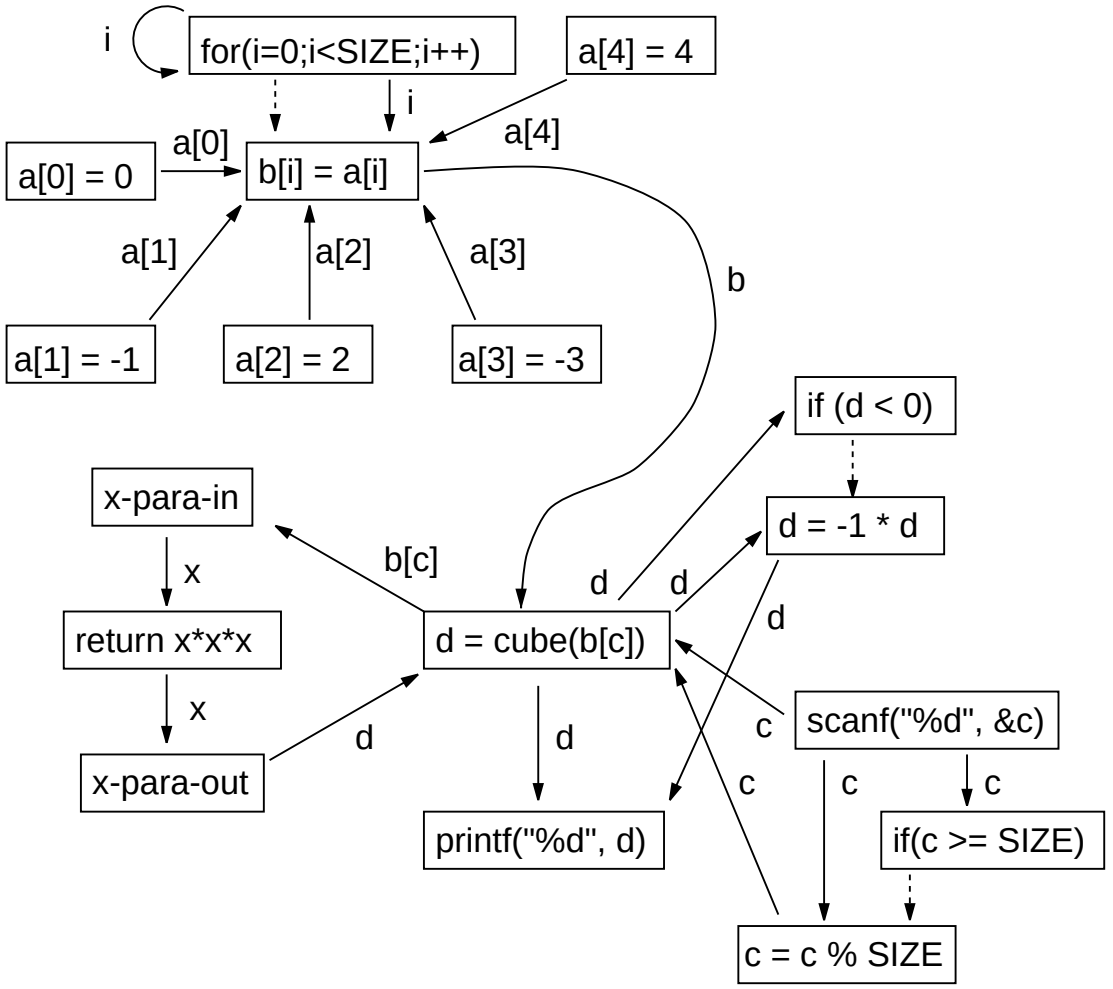


図 2: 図 1 のプログラムに対して静的に構築される PDG

サンプルプログラム

```
 1:  #include <stdio.h>
 2:  #define SIZE 5
 3:
 4:  int cube(int x) {
 5:      return x*x*x;
 6:  }
 7:
 8:  void main(void)
 9:  {
10:      int a[SIZE];
11:      int b[SIZE];
12:      int c, d, i;
13:
14:      a[0] = 0;
15:      a[1] = -1;
16:      a[2] = 2;
17:      a[3] = -3;
18:      a[4] = 4;
19:
20:      for (i=0;i<SIZE;i++) {
21:          b[i] = a[i];
22:      }
23:
24:      printf("Input: ");
25:      scanf("%d", &c);
26:
27:      if (c >= SIZE) {
28:          c = c % SIZE;
29:      }
30:
31:      d = cube(b[c]);
32:
33:      if (d < 0) {
34:          d = -1 * d;
35:      }
36:
37:      printf("%d\n", d);
38:  }
```

静的スライス結果

```
 1:  #include <stdio.h>
 2:  #define SIZE 5

 4:  int cube(int x) {
 5:      return x*x*x;
 6:  }

 8:  void main(void)
 9:  {
10:      int a[SIZE];
11:      int b[SIZE];
12:      int c, d, i;

14:      a[0] = 0;
15:      a[1] = -1;
16:      a[2] = 2;
17:      a[3] = -3;
18:      a[4] = 4;

20:      for (i=0;i<SIZE;i++) {
21:          b[i] = a[i];
22:      }


25:      scanf("%d", &c);

27:      if (c >= SIZE) {
28:          c = c % SIZE;
29:      }

31:      d = cube(b[c]);

33:      if (d < 0) {
34:          d = -1 * d;
35:      }

37:      printf("%d\n", d);
38:  }
```

図 3: サンプルプログラムのスライス基準を $(37, d)$ としたときの静的スライス

3　情報漏洩解析

3.1　情報漏洩解析

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 [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 [3]．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 [9] では，[3] 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，[16, 13] 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．

情報漏洩解析には，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 [16]
- 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 [3]

の 2 つの手法があるが，本稿では前者に着目する．

3.2 既存の情報漏洩解析手法

プログラムの入力値の SC から出力値の SC を求める手法として，既存の情報漏洩解析において計算に利用される手法は 2 種類の情報フローを利用して SC 方程式と呼ばれる再帰的な関係式を定義し，連立方程式の繰り返し計算を行なう方法である．

プログラムスライスについて考える．プログラムスライスでは，プログラムは PDG によって表され，プログラム内部の関係として依存関係が利用される．ここで利用される依存関係は，情報フローの場合と同じようにデータ依存・制御依存の 2 種類に分類でき，それぞれの性質も情報フローの場合と非常に似ている．また，セキュリティ解析において SC 方程式を定義する際には， プログラム中の各文で参照される各変数が変数がどの代入文によって定義された値を取り得るかを解析する．この解析は，プログラムスライスにおける PDG 構築時の依存関係解析と同じ手法で実現することが出来る．

このように，情報漏洩解析とスライスは非常に性質が似ている．そこで，情報漏洩解析の実現にはスライスにおける解析手法を応用できる．

既存の情報漏洩解析では PDG の構築手法を利用し，データフロー方程式を各文毎に抽出し，手続き毎に繰り返し計算を繰り返すことで，各入力において設定された SC がどのようにプログラム中の各変数を伝わって出力されるかを解析するという手法である．

PDG において情報フローを表す辺は，データ依存辺と制御依存辺の二種類で表現される．一方で，SC 方程式においては explicit flow と implicit flow の二種類で表現されている．このデータ依存辺と explicit flow，制御依存辺と implicit flow は，それぞれ同じ性質を持っているため，それぞれを対応させる．つまり，アルゴリズム中のデータ依存辺を構築する部分を，explicit flow の計算部に，制御依存辺を構築する部分を implicit flow の計算部に置き換え，解析を行なっている．

また，スライスの PDG の構築における手法では解析の各時点において各変数の値がどの場所で定義されうるかの情報が必要であるため，(変数, 値が定義された場所) の組の集合である到達定義集合を用意して解析を行なっていた．しかし，情報漏洩解析においては，解析に必要な情報は解析の各時点に各変数が持ちうる SC についての情報だけで十分であり，どの場所で定義されたかの情報は必要ない．そこで，解析中に参照される変数が持つ SC を表現する集合として，要素が (変数, SC) の組である**セキュリティクラス集合**（*Security Class Set*，以降，SCset と省略する）を定義する．解析では，プログラムの実行順に従い SCset を逐次更新することで出力文における SC を求める．

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．

3.2.1 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，SCset を用意する．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 4，図 5 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

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

の形式で記述している．図 6 には図 4，図 5 で利用される要素を示している．

3.2.2 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 [15] 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．

(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 $- kill \cup gen$

(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 $- kill \cup gen$

(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 B$_{then}$ else 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$_{pre}$:= $\emptyset$;

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 4: 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 5: 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

⊔**:** 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 6: アルゴリズムの要素

4　PDG を用いた情報漏洩解析

本節では，2，3 の内容をふまえ，既存の情報漏洩解析手法の問題点と，本研究で提案する新たな手法について述べる．

4.1　既存の情報漏洩解析手法の問題点

3 で述べた既存の情報漏洩解析手法はデータフロー方程式における繰り返し計算に基づいているため，手続きごとに繰り返し計算が必要となる．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うので，特に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．さらに，入力文の SC の初期値を変更するたびに再度繰り返し計算が必要となる．このため，大規模なプログラムにおいて，様々な SC の初期値を入力文に与えて解析を行う場合に時間的コストが問題となる．また，解析アルゴリズムを言語ごとに定める必要があるため汎用性に乏しい．

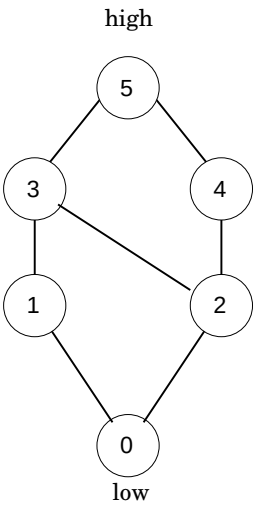
その他，SC を $high, low$ の二値のみしか実現していないため，複雑な束構造を持つ SC を対象とした情報漏洩解析を行うことができず，実用性に乏しい．

4.2　新たな情報漏洩解析手法の提案

既存の手法では 4.1 で述べたような問題点が考えられる．そこで我々は新たな情報漏洩解析の手法として，フォワードスライスを求めるアルゴリズムを利用し，PDG を探索することにより情報漏洩解析を行う手法を提案する．こうすることで，プログラムスライスにおいて利用している手法を採用することができ，実用性の高い手法となる．また，一般的な束構造を持つ SC を対象とした情報漏洩解析を行う．ここでは束構造の全ての 2 元の最小上界を二次元行列を用いて表すことで，一般的な束構造に対応する．

4.2.1　フォワードスライスと情報フロー

3.2 において述べたように，プログラムスライスと情報漏洩解析では共にプログラム内部の依存関係が利用され，プログラムスライスにおけるデータ依存関係と，制御依存関係は，情報フローにおける explicit flow，implicit flow に対応している．よってプログラムスライスの PDG におけるデータ依存辺と制御依存辺は，explicit flow と implicit flow をそれぞれ表しているので，ある入力文をスライス基準とした時のフォワードスライスは，その入力文において定義されている変数についての情報フローを表していると考えられる．よって，プログラム中の全ての入力文について，各々の入力文をスライス基準としてフォワードスライス抽



ハッセ図

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	1	3	3	5	5
2	2	3	2	3	4	5
3	3	3	3	3	5	5
4	4	5	4	5	4	5
5	5	5	5	5	5	5

束構造の和演算を表す二次元行列

図 7: 束構造の表現

出手法に基づいて PDG を探索しながら, 到着した各々の頂点で SC の和演算を行えば，最終的に全ての出力文の SC が求められる．

到着した各頂点での SC の和演算とは，その頂点の現在の SC と，スライス基準とした入力文で読み込まれる値の SC を入力として，その最小上界をとる演算のことである．(詳しくは 4.2.2，4.3.2 参照)

4.2.2 一般的な束構造を持つ SC を対象とした情報漏洩解析手法

一般的な束構造を持つ SC を対象とした情報漏洩解析は，これまで提案されてはきたものの実現はされていない．本研究では，ユーザがあらかじめ束構造を記述し，それを参照して SC の演算を行うことで一般的な束構造を持つセキュリティクラスに基づく情報漏洩解析を実現した．これにより，現実のモデルをより忠実に高速に表現することができる．具体的には以下の通りである．

束構造の表現

束構造の各元同士の最小上界を二次元行列で表す．また，各元同士の最小上界を求めることを，ここでは和演算と呼ぶ．

図 7 に，束構造と二次元行列の対応の例を示す．左のハッセ図で表される束構造が，右の二次元行列として表される．ハッセ図の頂点が束の元であり，解析対象のプログラムの SC

である．各 SC に数字を割り振り，最小元を low，最大元を $high$ と表す．この図では low は 0，$high$ は 5 と対応している．

右の二次元行列の m 行 n 列の要素が元 m と元 n の和を表す．例えば，元 1 と元 2 の和は元 3 であり，二次元行列の 1 行 2 列目の要素の値は 3 である．この二次元行列を参照することで，SC が a と b の和演算を求めることができる．また，この二次元行列から a と b の大小関係を計算できることを示す．a と b の最小上界を c とし，a と b の SC が等しいことを $a = b$，等しくないことを $a \neq b$，a が b より大きいことを $a > b$ と表すと，

- $a = b = c$ のとき $a = b$
- $a = c$,$b \neq c$ のとき $a > b$
- $b = c$,$a \neq c$ のとき $b > a$
- $a \neq c$,$b \neq c$ のとき a,b 間に大小関係なし

以上のように二次元行列から大小関係も計算できる．この表現方法を用いて一般的な束構造を持つ SC に基づく情報漏洩解析を行う．

4.3 PDG を利用した一般的な束構造に対する情報漏洩解析

4.2 で述べた方法の実現を考える．全ての頂点において SC の初期値は low とする．解析は次のような手順をたどる．

Phase 1: 依存関係解析 (データ依存関係，制御依存関係)

Phase 2: プログラム依存グラフ (*Program Dependence Graph,PDG*) の構築

Phase 3: 前提条件の入力

Phase 4: PDG の探索と SC の和演算

Phase1，Phase2 はそれぞれプログラムスライスにおける，2.1.1Phase 1:依存関係解析，2.1.2Phase 2:プログラム依存グラフの構築と同様である．

4.3.1 Phase 3:前提条件の入力

Phase3 では解析対象となるプログラムの 2 つの前提条件を入力する．

SC の束構造:

4.2.2 で述べた形で表現された，SC の束構造の定義を入力.

各入力文で読み込まれる値の SC:

各入力文について，そこで読み込まれる値の SC を入力.

4.3.2 Phase 4:PDG の探索と SC の和演算

Phase4 では PDG の探索をしながら，経路上の各頂点で SC の和演算を行う.

PDG の探索

PDG の探索は，解析対象のプログラムの各入力文とそこで定義される変数の組をスライス基準とし，explicit flow をデータ依存関係辺，implicit flow を制御依存関係辺で表していると読み換え，2 に示したフォワードスライス抽出における PDG 探索アルゴリズムに準じて探索を行う．スライス基準とした入力文で読み込まれる値の SC を sec とする．到着した頂点において，その頂点の現在の SC と sec を入力として SC の和演算を行い，頂点を演算結果の SC で更新する．次の依存関係辺をたどる前に，次の頂点が SC の和演算を実行して更新され得るかどうかをチェックして，更新されないならばその辺はたどらない．更新され得るならば辺をたどり次の頂点へ移る．SC が更新される頂点がなくなれば次の入力文をスライス基準として PDG 探索と SC 和演算を行う.

SC の和演算

各頂点では SC の和演算を行う．入力となるのは，

- スライス基準とする入力文で読み込まれる値の SC である sec
- 現在の頂点の SC である $vsec$

の二値である.

本研究の目的である「プログラムの入力値の SC から出力値が持つであろう SC を求める」ために，PDG を探索し，その経路上の各々の頂点で，情報フローの起点である入力文の SC である sec と，現在の頂点の SC である $vsec$ の和を新たにその頂点の SC とする．これにより，起点から現在の頂点にいたるまでの経路上の頂点の SC は適切な値となっている．以上を全ての入力文について繰り返すことで，不適切な情報漏洩を防ぐことができる.

4.4 解析の例

4.3 で示したアルゴリズムにより情報漏洩解析を行なった時の様子を図 8 に示す．図 8-1 が前提条件を入力した直後の PDG である．SC の束構造は図の右下に示す．入力文を表す頂点にセットされた SC は頂点の色の濃さで示す．起点とした入力文の SC が辺をたどって PDG の各頂点の SC を更新していく．もし次の頂点の SC が，起点とした入力文の SC より大き

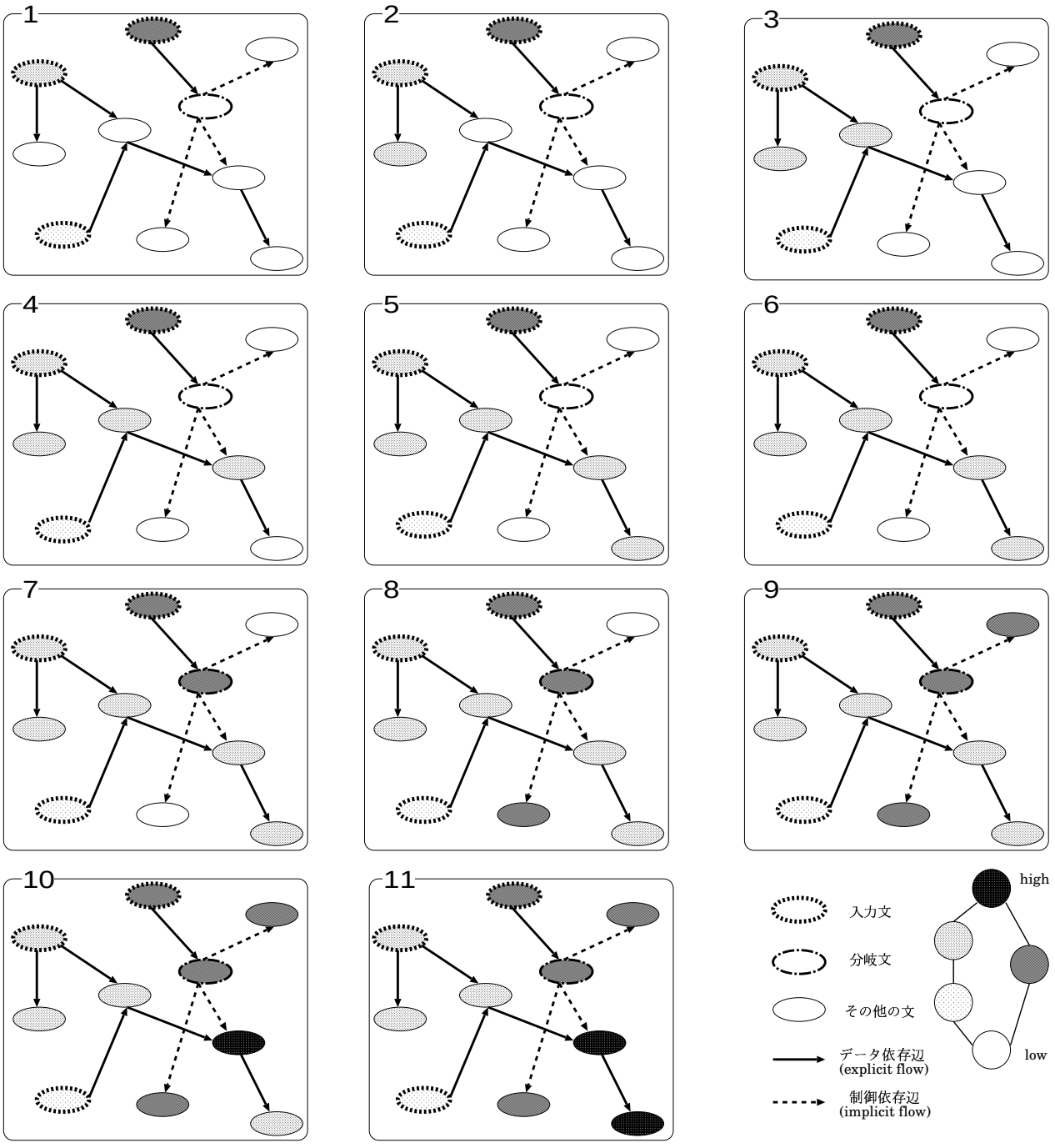

1
2
3
4
5
6
7
8
9
10
11
入力文
分岐文
その他の文
データ依存辺
(explicit flow)
制御依存辺
(implicit flow)
high
low

図 8: 解析の例

ければその辺はたどらない．たどる辺が無くなれば，次の入力文を起点として再び探索を行なう．図 8-11 が最終的な結果である．

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 “)”].

5 PDG を用いた情報漏洩解析システムの実現

本節では，本研究で実現した情報漏洩解析ツールの概要について述べる．

5.1 実装の方針

本ツールの実現は，我々が開発したスライスツールである *Osaka Slicing System*，*OSS*[6, 15] の静的スライス抽出部をもとにした 4 で述べた情報漏洩解析部を機能追加する形で行った．

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

5.2　SC 制約機能

情報漏洩解析の問題点として，SC の高いデータに対してプログラム中で暗号化やマージを行ってもとのデータが隠蔽された場合でも SC は高いままとなってしまうことが考えられる．実際に運用する際には，暗号化などが信頼できるとユーザが判断したときには，プログラム中の関数による出力やデータフローをユーザが任意にその SC を設定できる機能が必要である．

5.2.1　実現方法

今回の実装においてはその機能として，関数の戻り値の SC をユーザが任意に設定できるように実現した．手順としては前提条件の設定時の際，関数の戻り値を設定できる．具体的には，PDG 中のその関数の exit 頂点の SC に探索の際に与えられた SC をセットし，さらに以後の探索において，その頂点から情報フローするデータの SC をセットされた SC とする．以降はセットした SC を持つデータが情報フローすることになる．

この機能により，例えばプログラム中で関数の戻り値に与えられる SC の高いデータが関数内で隠蔽され，現実的にはその隠蔽されたデータからもとのデータを類推することが不可能であるとき，その値の SC を強制的に低くすることで，現実的な安全度に則した情報漏洩解析を行なうことができる．

5.3　ツールの解析の流れ

ツールの解析の流れを図 9 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 9-1）．次に，ユーザはソースファイル上で情報漏洩解析の前提条件を設定（図 9-2）し，UI 部を通じて情報漏洩解析部へ依頼する．情報漏洩解析部は，前提条件を基に解析を行ない（図 9-3）その結果を UI 部に渡す．UI 部は，ユーザが定義した各 SC の色を各々の文の背景色とすることで，各々の文の SC を表示する（図 9-4）．

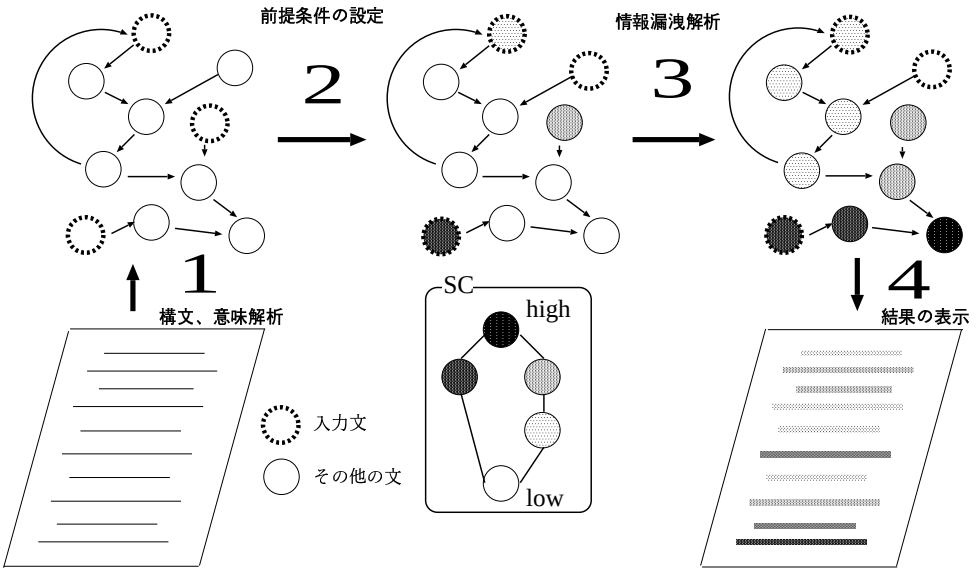


図 9: 解析の流れ

6 検証

本節では 5 で作成した情報漏洩解析アルゴリズムを実現した情報漏洩解析システムの具体的な適用事例を取り上げその有効性を検証する．6.1 で同条件で情報漏洩解析を行ったときの，既存のシステムとの実行時間の違いについて述べる．また，6.2 では複雑な束構造を SC として持つサンプルプログラムを考え，それに対してシステムの適用を行った．

6.1 前手法との実行時間の比較

4.1 で，既存の情報漏洩解析手法は手続き毎に必要な繰り返し計算のために，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効率が落ち，大規模な入力文の SC の初期値を変更するたびに再度繰り返し計算が必要なので，このため，大規模なプログラムにおいて，様々な SC の初期値を入力文に与えて解析を行う場合，時間的コストが問題となることを述べた．

既存のシステムと作成したシステムについて同条件で解析を繰り返し行ない，その実行時間の累計を比較する．図 10 に結果を示す．

図の通り，PDG 構築を行う必要がある一度目の解析においては実行時間にほとんど違いはないが，解析を繰り返し，回数を重ねるに従って既存のシステムと作成したシステムの実行時間の累計に開きが出てくる．既存のシステムは一度目の解析にかかる時間と以降の解析の時間が変わらない．しかし，作成したシステムでは二度目からの解析時間は減少している．

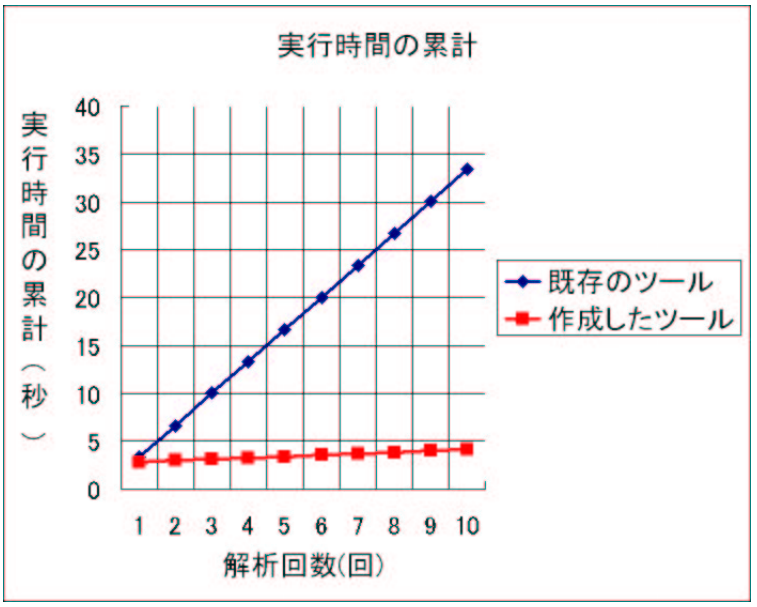


図 10: 実行時間の測定結果 (実験環境:Pentium4 2GHz，メモリ 1GB，Free-BSD4.5)

これは作成したシステムが一度構築した PDG の持つ情報を繰り返し利用できるためである．

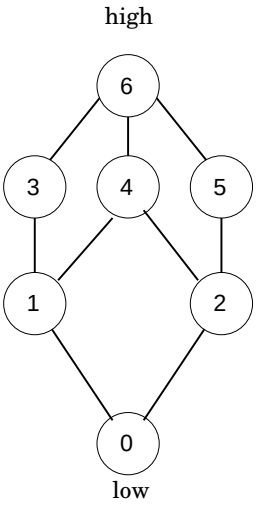
このことからプログラムが大規模になった場合，本手法は効率的である．

6.2 学生の成績管理システムに対する適用事例

次に複雑な束構造を SC として持つプログラムの安全性の確認に本システムを適用する．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．

学生の成績管理システムを考える．成績は一般教養科目分野と，専門科目分野に分類でき，それぞれ「可」または「不可」のいずれかである．全ての学生は自身の成績を，分野に限らず参照できる．また，一般教養科目分野，専門科目分野にはそれぞれ事務員が居り，一般教養科目分野の事務員は全ての学生の一般教養科目の成績の参照と書き換えを行うことができるが，専門科目分野の成績を参照，書き換えすることはできない．専門科目分野の事務員は，全ての学生の専門科目分野の成績の参照と書き換えを行うことができるが，一般教養科目分野の成績を参照，書き換えすることはできない．

学生，一般教養科目分野の事務員，専門科目分野の事務員はそれぞれ認証番号をもっており，成績管理システムにアクセスする際にそれぞれの認証番号を入力する．入力された認証番号によりシステムはユーザが何者かを判断し，それぞれのユーザに応じた権限を与える．



ハッセ図

図 11: システムの SC を表す束構造

そのユーザは以後その権限に応じた操作を行うことができる．

各ユーザの参照できるデータを整理すると，

学生: 自身の一般教養，専門科目分野の成績．自身の認証番号．

一般教養科目分野の事務員: 全ての学生の一般教養科目分野の成績．一般教養科目分野の事務員の認証番号．

専門科目分野の事務員: 全ての学生の専門科目分野の成績．専門科目分野の事務員の認証番号．

となる．

以上より，データとユーザから考えた，このシステムの SC の束構造を図 11 のハッセ図に示す．図 11 をもとに，それぞれのデータの SC と，ユーザのクリアランスの SC に対するマッピングを考える．主要なデータとその SC は，データ d の SC を $SC(d)$ と表すと，

一般教養科目分野の成績 $cgrade$**:** $SC(cgrade) = 1$

専門科目分野の成績 $sgrade$**:** $SC(sgrade) = 2$

一般教養科目分野の事務員の認証番号 $copcode$**:** $SC(copcode) = 3$

専門科目分野の事務員の認証番号 $sopcode$**:** $SC(sopcode) = 5$

学生の認証番号 $studentcode$**:** $SC(studentcode) = 4$

同様にユーザとそのクリアランスは，ユーザ u のクリアランスを $clear(u)$ と表すと，

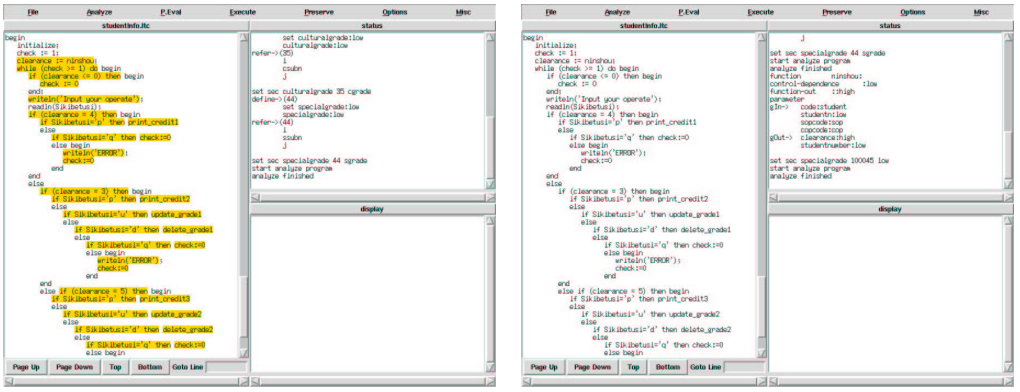
学生 $student$**:** $clear(student) = 4$

一般教養科目分野の事務員 cop**:** $clear(cop) = 3$

専門科目分野の事務員 sop**:** $clear(sop) = 5$

となる．

サンプルプログラムの各々のデータに対して上で述べた SC を与えて解析を行なった結果，33 個の出力文のうち，SC が 4 の出力文が 1 個，5 の出力文が 1 個，6 の出力文が 26 個という結果が得られた．(図 12-(a))



(a) (b)

図 12: 解析結果

これは，各ユーザの認証番号が様々な implicit flow を引き起こし，そのためユーザの認証を行なう関数が現在のユーザのクリアランスを判別した結果として返す値の SC が高くなり，その値が引き起こす implicit flow を受ける文が多数あったために起こった．しかし，各ユーザの認証番号や，関数が返す値が十分に隠蔽されたときを想定して，認証関数の戻り値の SC を low に設定して再度解析を行なうと，(図 12-(b)) 33 個の出力文のうち，SC が 0 の出力文が 27 個，1 の出力文が 2 個，2 の出力文が 2 個，4 の出力文が 1 個，5 の出力文が 1 個と，適切なデータに隠蔽を行なえば，現実的な情報漏洩の可能性が大きく下がることがわかる．このように，5.2 で述べた機能は，どのデータに隠蔽を施すべきかを判断する指針にもなる．

7　まとめ

本研究では，一般的な束構造を持つセキュリティクラスを対象とした，PDG を利用した情報漏洩解析を行なう手法を提案し，実現した．また，より現実的な利用を目指し，関数の戻り値をユーザが任意に設定できる機能を実装した．複雑な束構造を持つセキュリティクラスに基づいたデータベースシステムを試作し，実現したツールをそのプログラムに対して適用し，その有効性を確かめた．

今後の課題として，

- 情報フローの定義の拡張
- 他のプログラミング言語に対する情報漏洩解析アルゴリズムの拡張と実装

などが挙げられる．

謝辞

本研究において, 常に適切な御指導および御助言を頂きました大阪大学 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.

本研究において, 常に適切な御指導および御助言を頂きました 同 楠本 真二 助教授に深く感謝致します.

本研究において, 常に適切な御指導および御助言を頂きました 同 松下 誠 助手に深く感謝致します.

本研究において, 常に適切な御指導および御助言を頂きました 同 大畑 文明 氏に深く感謝致します.

本研究において, 常に適切な御指導および御助言を頂きました 同 横森 励士 氏に深く感謝致します.

最後に, その他様々な御指導, 御助言を頂いた大阪大学 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いたします.

参考文献

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

[2] T. Reps, S. Horwitz, M. Sagiv and G. Rosay, “Speeding up Slicing”, In Proceedings of the ACM SIGSOFT ’94 Symposium on the Foundations of Software Engineering, pp. 11–20, 1994.

[3] 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.

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

[5] S. Horwitz. and T. Reps. , “The Use of Program Dependence Graphs in Software Engineering”, Proceedings of the 14th International Conference on Software Engineering, pp. 392–411, 1992.

[6] Yoshiyuki Ashida, Fumiaki Ohata and Katsuro Inoue, “Slicing Methods Using Static and Dynamic Information”, Proceedings of the 6th Asia Pacific Software Engineering Conference (APSEC’99), Takamatsu, Japan, pp. 344–350, 1999.

[7] D.C. Atkinson and W.G. Griswold, “The design of whole-program analysis tools”, Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, pp. 16–27, 1996.

[8] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman, “Compilers : Principles, Techniques, and Tools”, Addison-Weseley, 1986.

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

[10] Harman, M. and S. Danicic, “Using program slicing to simplify testing”, Journal of Software Testing, Verification and Reliability, Vol. 5, No. 3, pp. 143–162, 1995.

[11] Binkley, D., Horwitz, S. and Reps, T., “Program integration for languages with procedure calls.”, ACM Transactions on Software Engineering and Methodology 4, Vol. 1, pp.3–35, 1995.

[12] K.B.Gallagher, “Using Program Slicing in Software maintenance”, IEEE Transactions on Software Engineering, Vol. 17, No. 8, pp. 751–761, 1991.

[13] Shigeta Kuninobu, Yoshiaki Takata, Hiroyuki Seki and Katsuro Inoue”, “An Efficient Information Flow Analysis of Recursive Programs based on a Lattice Model of Security Classes”, Proceedings of Third Internatinal Conference on Information and Communications Security (ICICS 2001), Xian, China, Lecture Notes in Computer Science 2229, pp. 292–303, 2001.

[14] 植田, 練, 井上, 鳥居: “再帰を含むプログラムのスライス計算法”, 電子情報通信学会論文誌, Vol. J78-D-I, No. 1, pp. 11–22, 1995.

[15] 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”, 情報処理学会論文誌, Vol. 37, No. 4, pp. 536–545, 1996.

[16] 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.

[17] 横森，大畑，高田，関，井上，“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”, 電子情報通信学会ソフトウェアサイエンス研究会,2000 年 11 月.