Vol.2015-DPS-162 No.13
Vol.2015-CSEC-68 No.13
2015/3/5

コンパイラを用いた情報フロー制御による情報漏洩防止機構

奥野 航平[1,a)]　内匠 真也[1]　大月 勇人[1]　瀧本 栄二[1]　毛利 公一[1]

概要：情報漏洩事件の多くは，人為的なミスを要因として発生していることが報告されている．そこで，本論文では，人為的なミスによる情報漏洩を防止するための機構 User-mode DF-Salvia を提案する．本機構は，情報フロー制御によってデータの利用方法を制限し，ユーザが意図しない情報の出力処理を制御することで情報漏洩を防止する．出力処理は，情報の源となったファイルに関連付いた保護ポリシに基づいて制御される．出力時の情報の保護ポリシを特定するためには，動的テイント解析を用いた情報フローの動的な追跡を利用する．これらのアクセス制御に必要な機能は，コンパイラによるコード変換を用いることでアプリケーションへ直接追加することによって実現し，プロセス単体でのアクセス制御を実現する．これにより，アプリケーションの置換え作業のみで本システムを導入でき，導入コストを削減できる．本機構の検証には，インターネット上の実アプリケーションを使用し，それらで情報漏洩が防止できることを確認した．

キーワード：アクセス制御，コード変換，情報フロー制御，動的テイント解析

A Data Loss Prevention System By Compiler and Information Flow Control

Kohei Okuno[1,a)]　Shin-ya Takumi[1]　Yuto Otsuki[1]　Eiji Takimoto[1]　Koichi Mouri[1]

Abstract: Many data loss incidents have reported to be caused by human error. This paper proposes a data loss prevention system caused by human error, that's called *User-mode DF-Salvia*. Our system breaks the procedures of human-mistaken data output and restricts data usage by information flow control. An output is controlled by a protection policy associated with a data source file. To get a protection policy when outputting the data, we use the dynamic information tracking technique as dynamic taint analysis. These features for access control are inserted to an application program by code transform in compiler. These applications make possible to control itself. Therefore, a user can simply deploy the access control system by replacement of an application and reduce deployment costs. Our system is evaluated by real applications published on the internet and we confirmed data loss prevetion.

Keywords: Access Control, Code Transform, Information Flow Control, Dynamic Taint Analysis

1. はじめに

情報システムの発展によって機密情報が電子化されるようになり，電子化された機密情報の漏洩事件が増加している．情報漏洩の主な原因として，JNSA 2013 年情報セキュリティインシデントに関する調査報告書 [1] では，次のものを挙げている．

[1] 立命館大学
Ritsumeikan University, Kusatsu, Shiga 525–8577, Japan
[a)] kokuno@asl.cs.ritsumei.ac.jp

アプリケーションの誤操作　ファイルを誤って電子メールに添付し，送信する．

データの管理ミス　ファイルを紛失（誤削除）する．ファイルが行方不明（誤移動）になる．

紛失・置忘れ　ファイルを USB フラッシュメモリなどの外部記憶装置へ書き込み，置忘れなどによって紛失する．

バグ　プログラムの誤った処理によってデータが流出する．

これらは，人為的ミスによって発生していると言える．また，情報漏洩の原因の約 8 割を占めており，大きな問題

© 2015 Information Processing Society of Japan

となっている．

情報漏洩を防止するための既存のセキュリティ技術として，ユーザ認証によるアクセス制御やファイルの暗号化などが存在する．しかし，これらの技術を利用した場合においても，一度プログラムに読み込まれたデータは利用方法に制限がないため，人為的ミスによる情報漏洩の防止は困難である．また，既存のアクセス制御を拡張したシステムとしては，SELinux [2] や TOMOYO Linux [3] などの強制アクセス制御がある．これらは，情報の伝達を制御する情報フロー制御によって，データの利用方法に対して制御できるため，人為的ミスによる情報漏洩の防止を可能とする．しかし，これらの情報フロー制御は，データ入出力の順序といった事前に定義された状態遷移に基づいて情報フローを推定するため，実際のプロセス内部の情報フローに従って制御されるとは限らない．また，従来のアクセス制御で使用されていた ACL やパーミッションなどのファイルに対して設定するポリシの他に，多種類のポリシを必要とするため，データの保護を目的とする設定の記述が複雑化しやすい．

以上の背景から，人為的ミスによる情報漏洩を防止する機構 User-mode DF-Salvia（以下，Salvia）を提案する．本機構は，ネットワークや外部記憶媒体など，プロセス外部へのデータ出力処理を制限するアクセス制御機構を持つ．本機構では，ユーザは，あらかじめ，保護したいファイルに対してファイルの利用方法を示したポリシ（保護ポリシ）を付加できる．アクセス制御機構は，プロセス内部の情報フローに基づいて出力されるデータからファイルに付いた保護ポリシを特定し，その保護ポリシに基づいて情報の伝播範囲を制限する．これにより，ファイル内の機密情報が外部に出力されることを防ぎ，人為的なミスによる情報漏洩を未然に防止できる．

情報フローは，あるオブジェクトが持つ情報が別のオブジェクトに伝わるときの流れである．一般に，計算機上では，ファイルやプロセスがオブジェクトとしてビット列であるデータを持つ．情報は，ビット列から人間が判断して読み取れる内容であり，様々な形式でデータとして保存される．そのため，単純なデータのコピーを追跡するだけでは，情報フローをすべて追跡することが難しく，特にデータ変換時やコピーを伴わない暗黙的な情報フローは，データに明確な情報が含まれることを保証できないため，制御漏れ（False Negative）や過剰な制御（False Positive）が発生する可能性がある．そこで，これらを解決するために，機械的に追跡できない情報フローをアノテーションによってプログラマが明示する方式を採る．また，情報フローには，プロセス実行時の制御フローによって動的に変化する特性があるため，動的に追跡することで実際の情報フローに基づいた制御を可能とする．

動的な情報フローの追跡を実現する方式として，ハードウェアを用いた方式 [4], [5]，バイナリ変換を用いた方式 [6]，コード変換を用いた方式 [7] など様々な方式が考えられる．ハードウェアを用いた方式では，特殊なハードウェアを必要とするため，導入のハードルが高い．また，ハードウェアを制御するための OS も必要となり，コストが大きい．バイナリ変換を用いた方式では，特殊なハードウェアを必要とせず，特定の実行可能ファイルのみを対象とできるため，導入が容易になる．しかし，機械語レベルで情報フローを追跡する必要があり，データに対して情報が含まれることを確認することが困難である．そこで，本機構は，コード変換を用いて単一プログラムに情報フロー追跡機能を付加し，特別な仕組みを必要としない方式を選択する．コード変換は，ソースプログラムレベルで情報フローを追跡することで，データに対して情報を持つフローの解析が容易になる．また，コード変換によって，情報漏洩の危険が想定されたプログラムのみに適用することも可能となる．これらは，アクセス制御の導入コスト削減，システム全体のスループット低下抑制，プラットフォームに依存しないアクセス制御の実現といったメリットがある．

以下，本論文では，2 章で Salvia について述べ，3 章でコード変換手法について述べる．4 章では，実アプリケーションを用いた評価としてアクセス制御機構の動作検証と性能評価について述べる．5 章では，関連研究について述べる．

2. User-mode DF-Salvia

2.1 データ保護を実現するアクセス制御

Salvia では，**図 1** に示すように，ユーザは，保護したいファイルに対して保護ポリシを付加できる．保護ポリシには，ファイルへの書出しの禁止（コピー禁止）やネットワークへの送信の禁止などのデータの利用方法の記述が可能である．計算機上のプロセスがファイルやネットワークへ情報を出力するときには，アクセス制御機構がデータの保護ポリシに基づいてその出力処理を制御する．したがって，ユーザは，事前にファイルに対して適切な保護ポリシを設定しておくことで，人為的ミスによる情報漏洩を防止できる．

2.2 アクセス制御の技術的な課題と解決策

2.1 節で述べたアクセス制御は，プロセスの情報フローに基づき，出力されようとしている情報の源となったファイルから保護ポリシを特定することで実現できる．しかし，情報は，人間がデータの内容を理解して得る内容であるため，データに含まれる情報量を機械的に判断することは困難である．そのため，データの変換を伴う情報フローや暗黙的な情報フローは，追跡が困難であり，追跡精度の低下を生じさせる．また，情報フローは，ユーザの入力操作などのコンテキストにより動的に変化するため，従来の

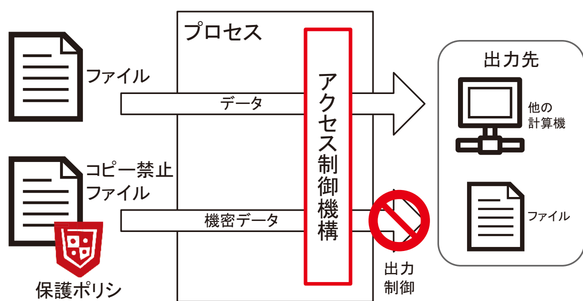

図 1 アクセス制御の概要

アクセス制御のようにデータを入出力する点（関数呼出しやシステムコール呼出し）のみでアクセス制御を実現することが困難である．そこで，Salvia は，この課題を動的テイント解析の応用とコンパイラによるコード変換を用いて解決する．

動的テイント解析は，プロセス実行時にデータに対して識別子（タグ）を割り当て，タグの伝播を追跡する技術である．動的テイント解析は，プロセス実行中にデータが別のデータ領域へ伝播したとき，タグを同時に伝播させる仕組みを持つ．この伝播処理を繰り返すことで，出力されるデータには，オリジナルデータと同一のタグが割り当てられるようになる．Salvia では，タグと保護ポリシを対応させておくことで，アクセス制御に使用する保護ポリシの識別を可能とする．

ただし，動的テイント解析による情報フローの追跡では，データがコピーされるような確実に情報が伝達しているフローのみを追跡の対象とする．これは，データに対して元々の情報が破壊される情報フローに対して，過剰な制御を抑制するためである．動的テイント解析の実現には，プログラムに対してデータの伝播が発生したときにタグを伝播させる処理を組み込む必要がある．Salvia は，コンパイラを用いてソースプログラムをコード変換し，必要な処理をソースプログラムに追加することで実現する．データの変換を伴う情報フローや暗黙的な情報フローは，アノテーションを用いたプログラマによる情報フローの明示によって追跡を可能とする．

Salvia は，コード変換を利用することにより，プログラム単体でアクセス制御が可能となり，プラットフォームに依存しないアクセス制御を実現する．そのため，本手法は，OS の再インストール無しでアクセス制御を導入できるというメリットがあり，既存の環境への導入が容易になる．

2.3 構成

Salvia の構成を**図 2** に示す．Salvia は，コード変換機構（Code Transform Module; CTM），アクセス制御機構（Access Control Module; ACM），変換ルール生成器（Transform Rule Generator; TRG）の 3 つのコンポーネントから構成される．

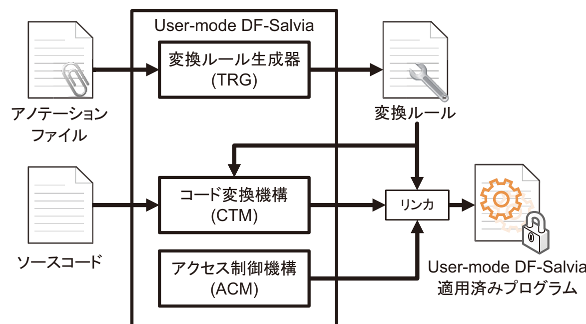

図 2 Salvia の構成

CTM は，プログラムのコードを変換するコンパイラであり，プログラムにアクセス制御に必要なコードを追加する．具体的には，LLVM [8] を用いて実装され，LLVM が提供する中間表現である LLVM IR を変換することでコードの追加を実現する．LLVM IR へは，フロントエンドを用いて各種プログラミング言語から変換できるため，CTM は，様々なプログラミング言語に対してアクセス制御の適用を可能とする．

ACM は，アクセス制御を行うためのライブラリである．ACM は，コード変換されたプログラムとリンクされたあと，プログラム実行時にアクセス制御を実現する．

TRG は，3.2 節で述べるアノテーションファイルを解析し，ライブラリ関数，データ変換，暗黙的フローの追跡のための変換ルールを生成するコンパイラである．変換ルールは，コード変換時にソースプログラムを変換するためのルールが定義されたファイルであり，CTM 内に組み込まれたパーサ，および，ACM 内に組み込まれたインタプリタによって処理される．

2.4 アクセス制御の方法

Salvia は，入出力処理を行う関数の実行を制御することでアクセス制御を実現する．関数の制御方法を**図 3** に示す．図 3 は，C 言語を用いたプログラムであり，ファイルにデータを出力する `fputs` 関数のアクセス制御の例を示している．図中の太字で示した部分がコード変換によって追加されたコードである．

CTM は，図 3 に示すように関数呼出しを対象にコード変換を行い，条件分岐を追加する．条件分岐は，ACM にてアクセス可否を判定し，その結果に基づいて関数を呼び出すか否かを決定する．この分岐によって，ライブラリ関数によるデータの入出力処理を禁止させることが可能となり，アクセス制御を実現できるようになる．

なお，ACM によりアクセスが拒否された場合は，関数の処理が失敗したときの状態を再現し，プログラムにアクセスが拒否されたことを通知する．具体的には，元の関数

© 2015 Information Processing Society of Japan

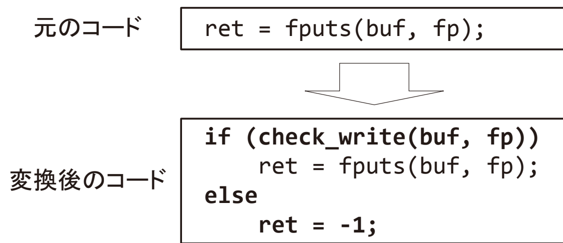

図 3　アクセス制御の方法

呼出しで戻り値を受け取っていた変数に対して，関数が失敗したときの戻り値を代入する処理を実行する．この処理は，プログラムを継続して動作させるために必要な処理であり，プログラムを失敗時の例外処理へ移行させるために必要となる．

2.5 情報フローの分類

Salvia では，データ変換による情報フローをデータの可逆性と情報の有無に着目して分類する．データの可逆性は，データの逆変換が存在し，かつ，変換前のビット列に完全に復元できることを指す．情報の有無は，変換後のデータに変換前の情報が含まれることを指す．情報フローをこれらの条件に基づき，以下の 4 つに分類する．

- 可逆（LOSSLESS）
  データの可逆性があり，かつ，情報を保持している情報フロー．例として，BASE64 などのエンコードがある．
- 暗号（CRYPT）
  追加情報を用いることでデータの可逆性を持ち，かつ，情報を保持している情報フロー．例として，AES, DES などの暗号処理がある．
- 情報保持（PRESERVE）
  データの可逆性は持たないが，情報を保持している情報フロー．例として，画像ファイルの不可逆圧縮や文字のレンダリング処理がある．
- 不可逆（LOSSY）
  情報が破壊される情報フロー．例として，MD5, SHA-1 などのハッシュ関数がある．

情報フローの追跡では，データに対して上記の情報フローの分類を識別できるように分類情報を含めたタグを割り当てて利用することで対応する．また，保護ポリシには，該当する分類のデータが出力されるときにどのように制御するかを指定できるようにしている．

情報フローの種類が複合された場合は，直前の分類と変換処理で指定された情報フローの分類を基に **表 1** に従って変換する．たとえば，直前の変換処理が可逆として分類さており，情報保持の変換処理を実行した場合，それらが複合された情報フローの分類は，情報保持として見なす．

表 1　データ変換行列

		変換処理の分類			
		可逆	暗号	情報保持	不可逆
直前の分類	可逆	可逆	暗号	情報保持	不可逆
	暗号	暗号	暗号	情報保持（1）	不可逆
	情報保持	情報保持	暗号（2）	情報保持（3）	不可逆
	不可逆	不可逆	不可逆	不可逆	不可逆

特筆すべき点として，（1）暗号から情報保持，（2）情報保持から暗号，（3）情報保持から情報保持への変換がある．（1）は，暗号化されたデータが破壊されるフローである．通常，ブロック暗号といった暗号方式では，データが欠損した場合，元のデータに復元することができない．そのため，この分類は，不可逆とすることも可能である．しかし，不可逆に分類した場合は，情報を一切持たないと判断するため，情報漏洩が発生する可能性がある．そのため，ここでは情報保持に分類している．

（2）は，一部が失われたデータが暗号化されるフローである．暗号は，可逆性が存在するフローを指すため，この組合せでは成立しなくなる．データの可逆性の観点では，このフローは情報保持として分類すべきである．しかし，実用上の観点では，このフローを情報保持として分類した場合，そのデータの外部出力が不可能になり利便性が低下する．そのため，ここでは暗号として分類している．

（3）は，データおよび情報が劣化するフローである．繰り返し変換が行われたとき，最終的には，不可逆なデータとなる場合がある．しかし，多くの場合で，データに多少なりとも情報が含まれるため，情報漏洩防止の観点からこのフローを情報保持として分類している．

3. コード変換

3.1 変換対象の関数

情報フローを追跡すべき関数で，次のものについては事前に列挙し，アノテーションに記述する必要がある．

- ファイルディスクリプタや FILE 構造体などのファイルを識別するための識別子（ファイル ID; FID）を作成・削除する関数
- FID を通じてファイル入出力を行う関数
- 情報フローが発生する関数

上記の関数は，ライブラリ関数とプログラマが定義した関数に分けられる．ライブラリ関数は，情報フローが発生する関数であっても CTM によってコンパイルされていないため，明示する必要がある．なお，これらの関数は，プログラミング言語やプログラミングインタフェースの仕様書などに記載されており，仕様からアノテーションを作成できる．また，あらかじめ，ひな形としてアノテーションを作成しておくことで，同一のライブラリ関数を使用する場合に再利用が可能となり，アノテーションを作成する手間を削減できる．プログラマが定義した関数については，データ変換や暗黙的情報フローが発生する場合に記述が必

© 2015 Information Processing Society of Japan

```
 1  typedef struct _IO_FILE FILE;
 2  typedef unsigned int size_t;
 3
 4  FILE *fopen(const char *path, const char *mode)
 5  {
 6          post (fopen != 0) {
 7                  open_file(fopen, path);
 8          }
 9  }
10
11  int fgets(char *s, int size, FILE *stream)
12  {
13          check {
14                  permit = check_read(stream);
15          }
16          fail {
17                  fgets = 0;
18          }
19          post (fgets != 0) {
20                  int len = strlen(s);
21                  tag_set(s, LOSSLESS, len, stream);
22          }
23  }
24
25  void *memcpy(void *dest, void *src, size_t n)
26  {
27          post (1) {
28                  tag_copy(dest, src, LOSSLESS, n);
29          }
30  }
31
32  void *memset(void *s, int c, size_t n)
33  {
34          post (1) {
35                  int i;
36                  for (i = 0; i < n; i++) {
37                          tag_copy(s + i, &c, LOSSLESS, 1);
38                  }
39          }
40  }
```

図 4 アノテーションファイルの例

要となる．

3.2 アノテーション

アノテーションは，C 言語をベースとした記述言語である．**図 4** に示すアノテーションファイルには，`fopen`，`fgets`，`memcpy`，`memset` のライブラリ関数に対するアノテーションが記述されている．

アノテーションは，それぞれの変換対象となる関数ごとに，`check`，`fail`，`post` の 3 種類のブロックに分かれている．それぞれのブロック内は，C 言語と同じ文法規則が利用できる．唯一，C 言語と異なる点は，アノテーションで指定された関数の戻り値を関数名で指定する点である．

`check` ブロックは，関数を実行する前に行う必要のある処理を記述する（13–15 行目）．このブロックには，暗黙的に 0 で初期化された変数 `permit` が宣言されており，この変数に対して 1 を設定することで，該当関数の実行が許可される．たとえば，`fgets` 関数の場合，ファイルの入力を許可することを確認する `check_read` 関数を呼び出し，アクセス可否の確認を行う (14 行目).

`fail` ブロックは，アクセス判定で拒否されたときに，関数のエラーをエミュレーションするために記述する（16–18 行目）．`check` ブロックが宣言されていた場合，このブロックは省略できず，必ず戻り値を設定する必要がある．`fgets` 関数の場合は，読取りエラーで戻り値として 0 を返すため，その処理を記述する（17 行目）．

`post` ブロックは，関数を実行した後に行う必要のある処理を記述する（6–8, 19–22, 27–29, 34–39 行目）．この処理は，`post` の後の括弧内で指定した条件を満たしたときのみ実行される．この条件には，通常，アノテーションで指定した関数が正常終了した時の条件を指定する．`fopen` 関数の場合は，正常にファイルを開いた後に `open_file` 関数を呼び出し，ファイルパスと FID を関連付ける処理を行う（7 行目）．`fgets` 関数の場合は，データ読出し後に `tag_set` 関数を呼び出し，データのアドレスに対してタグを設定する処理を行う（21 行目）．`memcpy` 関数，`memset` 関数の場合は，データコピー後に `tag_copy` 関数を呼び出し，タグを伝播させる処理を行う（28，37 行目）．

変数の型は，`typedef` を用いて宣言する（1–2 行目）．C++ 言語に代表される関数のオーバロードをサポートする言語では，関数名が引数の型や個数で名前修飾される．この名前修飾に対応するために，型宣言をアノテーションファイルに記述する．なお，`typedef` 宣言は，ライブラリ関数の定義がコンパイルする環境に依存するため，ライブラリ関数のアノテーションを記述する場合は，それぞれの環境に合わせて記述する必要がある．この記述は，コンパイル時にソースコードから型宣言を抽出することで自動化できる．

3.3 変換ルール

変換ルールは，TRG によってアノテーションファイルから生成されるオブジェクトファイルである．TRG は，アノテーションファイルを事前に解析することでアノテーションのエラーを検出し，また，オブジェクトファイルの生成によって実行時のパース時間を短縮する．

変換ルールは，独自の仮想的なスタック計算機を実行するためのプログラムを含み，アノテーションで指定された関数が呼び出されたときに実行される．

3.4 静的コード変換

Salvia の CTM では，アクセス制御を実現するために，以下の点をコード変換する．

- 代入処理
- 関数呼出し（引数・戻り値を通じた情報フロー）
- アノテーションに定義された関数の呼出し

3.4.1 代入処理

代入処理は，プログラム上の変数を別の変数にコピーする処理であり，情報フローが発生する．CTM は，代入処理の後にコードを追加し，実行時に ACM が代入元の変数のアドレスから代入先の変数のアドレスに対してタグを伝播させる．

代入処理に演算を含む場合は，データが変換されているため，情報が変わったと判断し，タグを伝播させない．また，定数でデータが上書きされた場合は，タグを削除する．

3.4.2 関数呼出し

関数呼出しでは，関数の引数が暗黙的にコピーされる．

```
1   char *acm_fgets(char *s, int size, FILE *stream)
2   {
3           int permit = check_read(stream);
4           if (permit) {
5                   ret = fgets(s, size, stream);
6                   if (ret != 0) {
7                           int len = strlen(s);
8                           taint_set(s, len, stream);
9                   }
10          } else {
11                  ret = 0;
12          }
13  }
```

図 5 fgets 用のラッパ関数

また，呼出し元に戻るときには，戻り値による情報フローが発生する．これらを追跡するために CTM は，関数を呼び出すコードの前後と関数の出入り口にコードを追加する．また，追加されたコードは，タグを以下の手順で伝播させる．

(1) 関数呼出し前に，実引数のアドレスとそのサイズを ACM に通知する．
(2) 関数の入口で，仮引数のアドレスとそのサイズを ACM に通知し，ACM が手順（1） で得た情報を基にタグをコピーする．
(3) 関数の出口で，戻り値として返される変数のアドレスとそのサイズを ACM に通知する．
(4) 関数呼出し元で戻り値が代入される前に，代入先のアドレスとそのサイズを ACM に通知し，ACM が手順（3）で得た情報を基にタグをコピーする．

3.4.3 アノテーションに基づくコード変換

CTM による静的なコード変換では，変換ルールの関数名をキーとしてソースプログラムを探索し，対象の関数呼出しが記述されているコードを変換する．検出した関数呼出しは，**図 5** に示すようなラッパ関数の呼出しにすべて置き換えられる．

図 5 は，図 4 の fgets に対するアノテーションから生成されたラッパ関数である．ラッパ関数とアノテーションは，3 行目と check ブロック，5–9 行目と post ブロック，11 行目と fail ブロックがそれぞれ対応している．

3.5 動的コード変換

間接参照を用いた関数呼出しでは，呼び出される関数が静的に決定しない．呼び出される関数を動的に取得するために，実行時に動的なコード変換を適用し，静的なコード変換と同様の機能を実現する．

CTM は，間接参照を用いた関数呼出しに対して，実行時にコード変換できるように**図 6** に示すようなコードに変換する．追加された関数は，アノテーションの動作と対応しており，check ブロックの動作が 図 6 の 1 行目，post ブロックの動作が 3 行目，fail ブロックの動作が 5 行目に対応している．ACM は，スタック計算機のインタプリタによって変換ルールを実行する．呼び出される関数の識別は，呼び出される関数のアドレスを用い，ACM の実行時に取得した変換ルールで定義された関数のアドレスとの比較によって実現する．また，変換ルールが存在しない場合は，3.4.2 項で述べた関数呼出しによる情報フローの追跡を適用する．

```
ret = pfunc(arg1, arg2, arg3);
```

⇩

```
1: if (dyn_prehook(pfunc, arg1, arg2, arg3)) {
2:   ret = pfunc(arg1, arg2, arg3);
3:   dyn_posthook(pfunc, ret, arg1, arg2, arg3);
4: } else {
5:   ret = dyn_error(pfunc);
6: }
```

図 6 間接参照を用いた関数呼出しの変換

4. 評価

本章では，Salvia の実アプリケーションに対する機能検証と性能評価について述べる．評価に用いた PC は，Intel Core i5-2320 の CPU と 16 GB のメモリを搭載し，OS として Fedora 20 x86_64 (Linux 3.17.7) が動作している．また，CTM は，LLVM 3.4 を使用して実装したものを使用している．

4.1 機能検証

機能検証では，以下のアプリケーションを用いて，情報漏洩が防止できることを確認する．以下のアプリケーションでは，可逆なデータ変換を伴う情報フローが存在し，それらの情報フローをアノテーションによって追跡できることを確認する．

- Mailx（メーラ）
  バイナリファイルの添付時に BASE64 を用いてエンコードされる処理が含まれており，データの変換が発生するフローが存在する．
- nkf（文字コード変換プログラム）
  異なる文字コードへ変換するときに，テーブルを使用したデータの変換を行うフローが発生する．

4.1.1 Mailx

Mailx を用いて情報フローを追跡できることを検証する．検証では，2 つのファイルを添付し，それぞれに関連付いた保護ポリシによってデータの出力が禁止されることを確認する．また，データが変換されるフローを追跡するために，C ライブラリ用のアノテーションに加えて，Mailx 用にアノテーションを追加で作成する．

Mailx 用の追加のアノテーションとして，Mailx のソースプログラムに定義されている BASE64 エンコード処理の関数 ctob64 を追加した．ctob64 関数は，エンコード対象の 3 文字が格納された文字列をエンコードした後，戻り値として変換後の文字列が格納されたアドレスを返す．この関数に対して，エンコード対象の文字列に付いたタグ

© 2015 Information Processing Society of Japan

を戻り値のアドレスに伝播させるアノテーションを記述した．追加したアノテーションの規模は，6 行である．

上記のアノテーションを用いて Mailx をコンパイルした後，ファイルを添付してメールを送信した．添付するファイルは，secret.png と public.png の 2 種類のバイナリファイルを使用した．それぞれの保護ポリシは，前者がコピー禁止，後者がコピー許可に設定している．

Mailx で送信したメールを確認したところ，2 つのファイルは，それぞれファイルとして添付されていることを確認した．しかし，secret.png は，ファイルの内容が空であり，かつ，エンコードされたデータの記述がメール本文中に無いことを確認した．一方，public.png は，添付したファイルと同一であることを確認した．この結果から，Salvia が保護ポリシに従ってデータ出力を制御したことを確認できる．ファイルに記述されたデータの出力のみが禁止されたため，空のファイルのみが添付されるという動作となった．また，エンコードされたデータは，Mailx 用に追加したアノテーションによって，データの変換を伴う情報フローの追跡が可能になり，制御が可能となっている．

4.1.2 nkf

本検証では，日本語で記述されたテキストファイルを異なる文字コードに変換し，テキストファイルに関連付いた保護ポリシによってデータの出力が禁止されることを確認する．前項の検証と同様に，nkf 用にアノテーションを作成した．

nkf 用の追加のアノテーションとしては，nkf のソースコードに記述されている `iconv` 関数群，`oconv` 関数群に対して作成した．`iconv` 関数は，特定文字コードから中間表現のような形式に変換する関数であり，文字コードを変換するときに必ず実行される関数である．この関数は，変換処理の後，`oconv` 関数を呼び出し，指定文字コードへ変換する処理が行われる．`oconv` 関数は，指定文字コードへ変換した後，標準出力へ出力する処理を行う．これらの関数の引数には，変換されていないデータが直接指定されるため，この関数が呼び出されるまでのフローは，自動的に追跡可能である．また，データの出力先は，標準出力に固定されているため，出力先は一意に識別できる．作成したアノテーションは，`iconv` 関数用と `oconv` 関数用の 2 種類であり，同系列の関数はコピー・アンド・ペーストを用いて作成している．追加したアノテーションの規模は，100 行程度であり，実質的な行数は，18 行である．

EUC-JP でエンコードされた日本語を含むテキストファイルを nkf で，UTF-8，SHIFT-JIS，JIS へそれぞれ変換したところ，ファイルへの出力時にすべて保護ポリシによって制御されることを確認した．また，同様に UTF-8 でエンコードされた日本語を含むテキストファイルを使用してそれぞれ変換した場合も，同様の結果を得られた．この結果から，エンコードされたデータ出力を制御できたことを確認できる．これは，Mailx の時と同様に，データの変換を伴う情報フローが追跡可能となったためである．

4.2 性能評価

4.2.1 方法

cp コマンドを使用してファイルをコピーするときのスループットを計測する．性能評価では，通常通りコンパイルしたプログラムと Salvia を使用してコンパイルしたプログラムを用意し，それぞれのスループットを比較する．

計測用にコピーするファイルとしては，合計サイズが 1 GiB になるようにランダムなデータが入った 8 個のファイルを RAM Disk 上に作成した．それぞれのファイルには，保護ポリシは付けず，ファイルがコピーされるようにする．計測では，2 種類の cp コマンドを用いて同一ファイルシステム上にコピーし，処理が完了するまでの時間からスループットを算出する．

4.2.2 結果と考察

通常通りコンパイルした cp コマンドのスループットが 2185 MiB/s，Salvia でコンパイルした cp コマンドのスループットが 1276 MiB/s となった．結果より，約 42 % の性能低下が表れた．

ファイルのコピー処理は，読出しと書込みが行われるため，これらが同一のスループットであると仮定すると，Salvia を適用した場合の読出し単体・書込み単体のスループットは 2552 MiB/s になると推定できる．このスループットは，PC で使用されるストレージの性能（Serial ATA 3.0 規格では最大でも 600 MiB/s 程度）と比較して，スループットに十分な余裕があることがいえる．また，性能低下の原因として，フロー追跡処理が考えられる．この処理は，CPU バウンドな処理であり，CPU の性能によって性能を向上させることができる．

5. 関連研究

5.1 動的テイント解析

動的テイント解析は，外部から攻撃の検知 [4] やマルウェアの解析 [5] など悪意あるデータを追跡するために使用されることがある．Argos [4] は，ゼロデイ攻撃を検知するためのエミュレータである．川古谷らは，マルウェアによって書き換えられたデータを追跡する手法 [5] を提案している．その他にも，libdft [6] や DTA++ [9] などの汎用的に動的テイント解析を利用できるようにしたフレームワークがある．これらの手法は，データに対して汚染（テイント）をマークしているが，そのデータが具体的にどのファイル，ネットワークアドレスから入力されたかを区別しない．Salvia は，データからファイルの源を特定し，それぞれに応じた制御を実現する．

一般に，暗黙的な情報フローが発生すると，タグの伝播が発生しないため情報フローを追跡できない [10]．そのた

© 2015 Information Processing Society of Japan

め，意図的に暗黙的な情報フローを発生させた場合，タグが伝播しない問題が発生する．川古谷らは，マルウェアのコードが書き換えたデータに対してもタグを付けることで問題を解決している．DTA++ では，事前にプログラムを解析することで，暗黙的な情報フローの追跡を実現している．Salvia では，アノテーションを用いて暗黙的な情報フローの追跡を可能とし，情報フローによって出力されたデータに対して情報が含まれているかどうかを分類して追跡する．そのため，暗黙的な情報フローが発生したときでも，アノテーションを適切に定義することで，過剰な制御を抑制することができる．また，アノテーションは，ソースプログラムに基づいて作成するため，幅広い種類の情報フローの追跡を可能とさせている．

5.2 Usage Control

既存のセキュリティモデルとして，Usage Control (UCON) [11] が提案されている．UCON は，データへのアクセス許可後の利用を制御するセキュリティモデルであり，情報漏洩の防止や知的財産の管理（DRM）などに適用が可能である．UCON で使用されるポリシとしては，「コピー禁止」，「30 日後にファイルを削除」，「5 回まで動画像を利用可能」などが利用できる．システムの実装としては，VMM と OS を使用した方式 [12] や Android 上に実装したシステム [13] などがある．

本手法では，情報漏洩の防止をターゲットとし，ポリシには，プロセスの動作を記述する方式を採用している．ポリシは，情報フローの追跡によってデータに対応付けられるため，データに応じてプロセスの動作を制限できる．ただし，プロセスが動作していないときのデータ利用は制限できず，主として DRM の用途において，自動的にファイルを削除するといった受動的なイベントには本システムで対応できない．しかし，人為的ミスによる情報漏洩は，プロセスが動作している時に発生するため，プロセスの能動的な動作を制限するだけで十分な対策ができると言える．

6. おわりに

本論文では，人為的なミスによる情報漏洩を防止する User-mode DF-Salvia について述べた．Salvia は，データの利用方法をユーザが保護ポリシで指定でき，情報フローに基づいてデータの利用方法を制限することで，情報漏洩の防止を可能とする．また，コンパイラを用いたコード変換により，情報フローの追跡，および，アクセス制御をプログラム単体で実現した．コード変換を実現したことで，既存のアプリケーションをコンパイルするだけでアクセス制御を導入きるメリットがある．評価では，実アプリケーションとして Mailx と nkf を用いて動作を検証した．Salvia による性能差は，適用していない状態と比較して約 42 % に低下したが，一般的なファイル入出力において十分なスループットを確認した．

今後の課題としては，アノテーションの記述漏れを防ぐようにするための手法の考案や，実行時のオーバヘッド削減用の静的解析などがある．

参考文献

[1] NPO 日本ネットワークセキュリティ協会 (JNSA)：JNSA 2013 年情報セキュリティインシデントに関する調査報告書，http://www.jnsa.org/result/incident/ (2015).

[2] Loscocco, P. and Smalley, S.: Integrating Flexible Support for Security Policies into the Linux Operating System, *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pp. 29–42 (2001).

[3] 原田季栄，保理江高志，田中一男：TOMOYO Linux - タスク構造体の拡張によるセキュリティ強化 Linux，*Proceedings of the Linux Conference 2004*, pp. 1–10 (2004).

[4] Portokalidis, G., Slowinska, A. and Bos, H.: Argos: an Emulator for Fingerprinting Zero-Day Attacks for advertised honeypots with automatic signature generation, *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pp. 15–27 (2006).

[5] 川古谷裕平，岩村　誠，針生剛男：テイント伝搬に基づく解析対象コードの追跡方法，情報処理学会論文誌，Vol. 54, No. 8, pp. 2079–2089 (2013).

[6] Kemerlis, V. P., Portokalidis, G., Jee, K. and Keromytis, A. D.: Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems, *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments 2012*, pp. 121–132 (2012).

[7] Chang, W., Streiff, B. and Lin, C.: Efficient and Extensible Security Enforcement Using Dynamic Data Flow Analysis, *Proceedings of the 15th ACM Conference on Computer and Communications Security 2008*, pp. 39–50 (2008).

[8] Lattner, C. and Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, pp. 77–86 (2004).

[9] McCamant, M. G. K. S. and Song, P. P. D.: DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation, *Proceedings of the 18th Annual Network and Distributed System Security Symposium* (2011).

[10] Cavallaro, L., Saxena, P. and Sekar, R.: On the Limits of Information Flow Techniques for Malware Analysis and Containment, *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Vol. 5352, pp. 143–163 (2008).

[11] Park, J. and Sandhu, R.: The UCONABC Usage Control Model, *ACM Transactions on Information and System Security*, Vol. 7, No. 1, pp. 128–174 (2004).

[12] Xu, M., Jiang, X., Sandhu, R. and Zhang, X.: Towards a VMM-based Usage Control Framework for OS Kernel Integrity Protection, *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies 2007*, pp. 71–80 (2007).

[13] Bai, G., Gu, L., Feng, T., Guo, Y. and Chen, X.: Context-Aware Usage Control for Android, *Security and Privacy in Communication Networks*, Vol. 50, pp. 326–343 (2010).

© 2015 Information Processing Society of Japan