Vol.2015-DPS-162 No.12
Vol.2015-CSEC-68 No.12
2015/3/5

コンパイラとOSの連携によるデータフロー追跡手法

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

概要：情報漏洩の多くは人為的なミスにより発生している．そこで，人為的ミスによる情報漏洩を防止するために，ファイルごとに設定可能なデータの機密度に基づいて，データの出力処理を制御するセキュアシステム DF-Salvia の開発を行っている．DF-Salvia では，コンパイラと OS が連携し，プロセス内部のデータフローを追跡する．本論文では，そのデータフロー追跡手法について述べる．具体的には，コンパイラによってデータフローの静的解析情報を生成するとともに，実行時に動的解析を可能とするためのデータフロー追跡用コードを挿入する．OS は，それらの情報をもとに動的にデータフローを解析する．本手法をアプリケーションに適用させた結果，データフローを追跡し，情報漏洩を防止できることを確認した．

キーワード：アクセス制御，データフロー解析，プログラム解析

A Data Flow Tracking Method Collaborated by Compiler and OS

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

Abstract: Many information leakage incidents are caused by human error. To prevent from these information leakage, we develop DF-Salvia that prevents an output based on a policy whom users set to a file. DF-Salvia tracks data flow inside a process by compiler and OS. In this paper, we propose this data flow tracking method. Compiler creates data flow information and inserts data flow tracking code to source code for analyzing dynamic data flow. OS analyzes data flow in run-time according to them. In the results of applying this method to applications, we conformed what DF-Salvia can prevent from information leakage by tracking data flow.

Keywords: Access Control，Data Flow Analysis，Program Analysis

1. はじめに

個人情報が，電子化され容易に扱えるようになったため，漏洩する危険性が高まっている．個人情報の漏洩は，プライバシの侵害となり，漏洩した企業のイメージダウンにつながるため，企業は情報管理に責任を持つ必要がある．情報漏洩に関する報告書 [1] によると情報漏洩の約 80 % は，「誤操作」や「管理ミス」，「紛失・置き忘れ」などの人為的ミスにより発生している．具体的な例として，「誤操作」では機密情報を添付したメールを間違った顧客に送信すること，「管理ミス」，「紛失・置き忘れ」では機密情報を保存した外部記憶装置を紛失することが挙げられる．これらの情報漏洩は，正当な権限を持つユーザにより引き起こされるという特徴がある．情報漏洩を防止する手段の一つとして，オペレーティングシステム (以下，OS) によるファイルアクセス制御機能を利用する方法が挙げられる．しかし，ファイルアクセス制御は，ファイルへの不正なアクセスを防止できるが，アクセスを許可されているユーザによる情報漏洩を防止することはできない．また，OS の中には強制アクセス制御機能を持つ Security-Enhanced Linux (SELinux) [2] や TOMOYO Linux [3] を組み込んだセキュア OS がある．セキュア OS では，ユーザやファイル，プロセスなどに対して権限を詳細に設定できるため，より強力なアクセス制御を行うことができる．しかし，強制アクセス制御では，複数のファイルを扱うプロセスに対して，

[1] 立命館大学 Ritsumeikan University,
525-8577, 滋賀県草津市野路東 1-1-1
[a] stakumi@asl.cs.ritsumei.ac.jp

© 2015 Information Processing Society of Japan

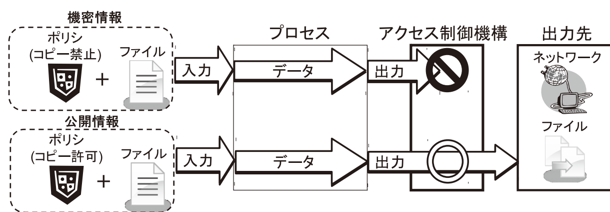

図 1 DF-Salvia によるアクセス制御

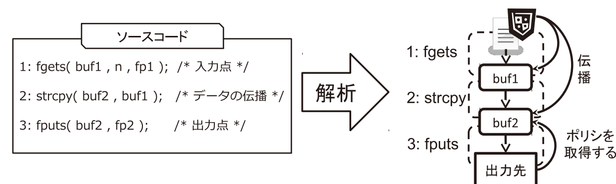

図 2 データフロー情報

機密情報を含んだ特定のファイルから読み込まれたデータの出力のみを防止することができない．そのため，必要以上にデータの出力が制限されてしまう False Positive が発生する．

以上の背景より，正当なアクセス権限を持つユーザによる情報漏洩を防止するセキュアシステム DF-Salvia [4] の開発を行っている．DF-Salvia は，ファイル単位でデータ保護ポリシ (以下，ポリシ) を設定可能とし，そのポリシに従ってデータの流れ (以下，データフロー) を制御することで情報漏洩の防止を実現する．アクセス制御は，出力されるデータの源となったファイルのポリシに基づいて行われる．このアクセス制御を実現するために，DF-Salvia は，プロセス内部のデータフローを追跡することでデータの源を特定する．

DF-Salvia は，コンパイラと OS の連携により上記のアクセス制御を実現し，システム全体に対して統一的なアクセス制御の機能を提供する．一般的に，OS は，プロセス単位で資源を管理するため，プロセス内部の細かなデータフローを追跡できない．また，OS は，プロセス上のメモリのデータ構造を把握することができないため，データが持つ意味を理解した上でそれを識別することが困難である．この課題を解決するために，プロセスの元となるソースコードに着目し，コンパイラを用いてソースコードのデータフローを解析し，OS はこの解析結果を用いてデータフローを追跡する．データ構造や制御構造が明示された高水準なプログラミング言語でデータフローを解析できるため，意味のあるデータを単位とした情報漏洩の防止を実現できる．

以下，2 章で DF-Salvia について述べ，3 章でデータフローの追跡手法，4 章で DF-Salvia の構成について述べる．5 章では，データフロー追跡手法の実装について，6 章では評価について述べる．次に，7 章で関連研究について述べ，8 章でまとめる．

2. DF-Salvia

DF-Salvia は，「どの」データが「どこに」出力されるかを示した情報を基にアクセス制御を実現する．DF-Salvia によるアクセス制御を**図 1** に示す．

DF-Salvia は，ユーザが保護対象ファイルに設定したポリシに基づいてファイルのデータの出力可否を判定し，情報漏洩を防止する．ポリシには，ファイルの出力先ごとに出力の可否を設定できる．例として，図 1 では，機密情報に全てのコピーを禁止するポリシを設定し，公開情報にコピーを許可するポリシを設定している．他にも，「USB メモリへのコピーは禁止する」,「特定の IP アドレスへの送信を禁止する」といったポリシを記述することもできる．

図 1 の例では，プロセスが機密情報のデータを出力しようとした場合，データの源となる機密情報のポリシに従ってデータの出力を禁止する．同様に，プロセスが公開情報のデータを出力しようとした場合，データの源となる公開情報のポリシに従ってデータの出力を許可する．

上記のアクセス制御では，出力されるデータの源を特定するためにプロセス内部で発生するデータフローを追跡する必要がある．

3. データフロー追跡手法

3.1 概要

データフローを追跡するためには，メモリに格納されるデータの源を識別する必要がある．そのために，DF-Salvia は，C 言語のソースコード中にあるファイル入出力，ネットワーク入出力，変数アクセスなどの命令文について，「どの」データが「どこに」格納されたかを解析する．さらに，解析結果に基づいて変数とその変数に格納されるデータのポリシを関連付けることでデータフローを追跡する．

上記のデータフローの追跡を行うことにより，ファイルのデータを格納する点 (入力点) とデータをプロセスの外部に出力する点 (出力点) を関連付け，「どこから」入力されたデータが「どこに」出力しようとしているか識別する．DF-Salvia が解析するデータフロー情報の例を**図 2** に示す．図 2 に記録されるデータフロー情報を行番号ごとに示す．

(1) 入力点である fgets 関数において変数 buf1 にファイルのデータが書き込まれる．
(2) 代入を発生させる strcpy 関数において変数 buf1 のデータが buf2 にコピーされる．
(3) 出力点である fputs 関数において変数 buf2 のデータが出力される．

解析された情報を基にした DF-Salvia によるデータフローの追跡処理を行ごとに示す．

(1) ファイルのポリシを変数 buf1 に関連付ける．
(2) 変数 buf1 に関連付けられたポリシを変数 buf2 に関連付ける
(3) 変数 buf2 に関連付けられたポリシに従って出力の可

© 2015 Information Processing Society of Japan

否を判定する．

上記の処理により，変数 buf1 と buf2 によって入力点と出力点が関連付けられるため，ポリシに従ったアクセス制御が可能となる．

3.2 型に基づくデータフロー追跡の粒度

C 言語では，データの種類が変数の型によって決定する．型は，基本型と複合型に大別される．基本型は，データを表現するための最小単位の型である．例として，int 型，char 型，アドレスを格納するポインタがある．複合型は，ユーザがソースコード中に定義した型で，複数の変数から構成される．例として，配列，構造体などがある．

基本型ごとにデータを識別することで，データを意味のある単位でデータフローを追跡できる．一方，複合型で定義された変数は，複数の変数から構成されるため，要素ごとに分割・識別してデータフローを追跡する必要がある．これは，複合型で定義された変数の先頭からの相対アドレス (以下，オフセット) と変数のサイズにより要素を識別することで実現する．ただし，char 型配列については，文字列やバイト列を格納することが多いため，要素を識別せず配列を 1 つの変数として扱う．

3.3 ポインタにおける参照先の記録

ポインタは，変数のアドレスを保持し，保持したアドレスのデータにアクセスする手段を提供する．すなわち，ポインタによって異なる変数が同じデータにアクセスできる別名関係が発生する．ポインタを用いてデータにアクセスする場合，変数にポリシを関連付ける手法では，ポインタと参照先変数に異なるポリシが関連付けられてしまう．これにより，誤ったポリシによって間違った制御が発生する．そのため，DF-Salvia は，実行時にポインタの参照先を記録し，ポインタによりアクセスされる変数を特定可能にする．

また，ポインタは，ポインタのアドレスを保持することで，参照をネストさせることができる．ネストされたポインタを使用した場合，ポインタの参照の深さ (以下，参照階層) によってアクセス先を変更できるため，参照階層を記録する必要がある．そのため，DF-Salvia は，あらかじめソースコードからポインタを用いたデータアクセス時の参照階層を記録する．これにより，DF-Salvia は，記録したポインタの参照先とデータアクセス時の参照階層を基にアクセス先の変数を特定可能にする．

3.4 動的な型変換による領域の再分割

C 言語では，型変換 (キャスト) によって変数を再分割し，分割された変数にそれぞれ異なるデータを格納できる．そのため，分割された変数に格納されるデータを識別するために，キャストに応じて変数を分割する必要がある．

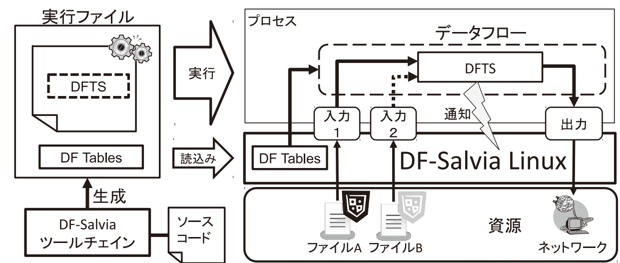

図 3 DF-Salvia の構成

キャストされた変数には，新しくデータが格納されるまでキャスト前のデータが格納されている．そのため，DF-Salvia ではキャストされた変数にデータが格納されたときに変数を分割する．

4. 構成

4.1 概要

DF-Salvia は，ソースコードを解析するツールチェインと OS から構成される．DF-Salvia の構成を**図 3** に示す．DF-Salvia ツールチェインは，DF-Salvia 独自のコンパイラとコマンドから構成され，OS は，Linux を改良した DF-Salvia Linux を用いる．

DF-Salvia ツールチェインは，C 言語で記述されたソースコードを解析し，データフロー情報を記録した DF Tables を生成する．同時に，プログラムの実行時に DF-Salvia Linux と連携するために，独自のシステムコールであるデータフロートラッキングシステムコール (以下，DFTS) の呼出しをプログラムに対して挿入する．

実行時に，プロセスは，DFTS によりデータフローの発生を DF-Salvia Linux に通知する．DF-Salvia Linux は，DFTS により受け取った通知と DF Tables を基にデータフローを追跡する．また，プロセスがデータを出力しようとするとき，DF-Salvia Linux は，そのデータのポリシに従って出力の可否を判定し，アクセス制御を行う．

4.2 ソースコードの解析

DF-Salvia ツールチェインのコンパイラは，変数を読書きする命令文について，それぞれデータを読み出す変数とデータを書き込む変数を DF Tables に記録する．DF Tables を構成する 5 つのテーブルについて，以下に述べる．

- Type Table
  型情報を記録したテーブル．このテーブルから取得した型情報により，実行時に領域の再分割を可能にする．
- Variable Table
  ソースコード中の変数を記録したテーブル．変数を識別するために用いる．
- Access Table
  命令ごとに，複合型の要素の特定やポインタの参照先の特定などのデータ構造に対するアクセス方法 (アク

セス情報) を記録したテーブル．命令がデータ構造のどこにアクセスするかを特定するために用いる．

- Def-Use Table
  ソースコード中の命令により読書きされる変数を特定可能にするテーブル．このテーブルにより，命令ごとに発生するデータフローが特定可能となる．また，読書きされる変数を特定するために，Variable Table と Access Table を用いる．Variable Table により命令に用いられる変数を特定し，その変数を起点とするデータ構造へのアクセス先を Access Table から特定する．
- Function Frame Table
  関数ごとのローカル変数を記録したテーブル．関数呼出しにより生成される変数を特定するために用いる．

4.3 DF Tables

DF-Salvia Linux は，動的に決定される情報を DFTS を通じて受け取り，データフローを追跡する．DF-Salvia ツールチェインのコンパイラは, 以下の処理に対して DFTS の呼出しを挿入する．

(1) 変数間における値の伝播
(2) 変数として確保される領域の生成
(3) 変数として確保される領域の削除
(4) 連続するデータに対するアクセス先の動的な決定
(5) 定数値の代入

(1) は，変数の値を代入する処理，関数の引数・戻り値，データの伝播が発生するライブラリ関数 (memcpy 関数，atoi 関数など) による処理である．これらの処理でデータがコピーされた場合，OS は，データを取り出す変数のポリシがデータの格納先変数に伝播したことを記録する．また，ポインタへの代入が発生した場合，OS は，ポインタが指す参照先を記録する．たとえば，int 型の変数 a と int 型の変数 b があり，a = b; といった命令が存在する場合，DFTS によりその命令の実行が OS に通知される．この通知から OS は，Def-Use Table を用いて左辺の変数と右辺の変数を特定する．さらに，a = b; はデータがコピーされるため，特定した変数を用いて，左辺の変数に関連付けられたポリシを右辺の変数に関連付ける．

(2) は，ローカル変数の確保，ライブラリ関数 (malloc 関数，calloc 関数など) によるメモリの確保，可変長配列による変数が確保される処理である．これらの処理により変数が生成された場合，OS は，生成された変数に対してデータフローの追跡を開始する．たとえば，malloc(4); の呼出しが存在した場合，DFTS によりサイズ 4 のメモリが確保されたことが OS に通知される．これにより，OS は，生成されたサイズ 4 の変数を Variable Table に登録し，データフローを追跡できるようにする．

また，たとえば，ローカル変数 a，b を持つ関数 func を呼び出された場合は，DFTS により関数 func が呼び出されたことが OS に通知される．この通知により，OS は，Function Frame Table から関数 func が生成するローカル変数 a，b を特定する．さらに，特定したローカル変数 a，b に対して，データフローを追跡できるようにする．具体的には，変数にポリシを関連付けられるようにし，ポインタの参照先を記録できるようにする．

(3) は，関数のリターン時におけるローカル変数の削除，free 関数による処理である．これらの処理が発生した場合，OS は削除された変数に対するデータフローの追跡を終了する．たとえば，ローカル変数 a，b を持つ関数 func からリターンする場合，DFTS により関数 func からリターンすることが OS に通知される．この通知により，OS は，Function Frame Table から関数 func が呼出し時に生成したローカル変数 a，b を特定する．特定したローカル変数 a，b に対して，データフローの追跡を終了する．

(4) は，配列やポインタに対してインデックスを変数で指定することで，アクセスする要素を動的に変更する処理である．そのため，OS は，インデックスの値を DFTS によって受け取り，アクセスする要素を特定可能にする．たとえば，a = b[i]; といった命令が存在した場合，DFTS により変数 i の値とその値を使用する命令が OS に通知される．前者の情報は，アクセス情報を書き換えるために使用され，後者の情報は，Access Table に記録されたアクセス情報を特定するために使用される．OS は，特定したアクセス情報に定義された複合型の要素を特定するためのオフセットを変数 i の値を用いて書き換える．その後，書き換えたアクセス情報は，a = b[i]; に対して挿入された DFTS がデータ構造へのアクセス先を特定するために利用される．

(5) は，変数に定数値を代入する処理である．定数値は，ファイルから読み込まれたデータではないため，OS は，定数値が代入された変数から関連付けられているポリシを外す．たとえば，a = 0 といった命令があった場合，DFTS によりその命令の実行が OS に通知される．この通知により，OS は，Def-Use Table から定数値を格納する変数を特定する．その後，その変数に関連付けられたポリシを削除する．

5. 実装

DF-Salvia ツールチェインのコンパイラは, コンパイラ基盤 LLVM [5] を用いて実装し，DF-Salvia Linux を，Linux 3.9.9 を改良して実装した．以下に，DF-Salvia ツールチェインと DF-Salvia Linux の実装について述べる．

5.1 DF-Salvia ツールチェイン

DF-Salvia ツールチェインは，コード中に DFTS の呼出しを挿入した実行ファイルを生成し，実行ファイルにソースコードから解析・生成した DF Tables を組み込む．そのために，LLVM に対してデータフローを解析する機能と

© 2015 Information Processing Society of Japan

DFTS の挿入を行う機能を実装した．

また，DF Tables の Def-Use Table には，実行時にどの地点で発生したデータフローかを特定可能にするためにデータフローが発生した命令アドレスを記録する．これにより，OS は，そのアドレスを基に Def-Use Table から発生するデータフローを特定することができる．しかし，アドレスはリンク時に決定するため，LLVM IR の解析時には，データフローの発生した命令アドレスを取得できない．そのため，DF-Salvia の独自コマンドは，実行ファイルからアドレスを取得し，取得したアドレスとデータフローの解析の結果を結合し，DF Tables を生成する．

DF-Salvia ツールチェインは以下に示す処理を行う．

(1) Clang を用いて，C 言語のソースコードから LLVM IR を生成する．Clang は，C 言語を LLVM IR に変換するフロントエンドである．
(2) LLVM IR からデータフローを解析し，解析結果を生成する．また，同時に DFTS を呼び出すためのコードを挿入する．
(3) LLVM IR から LLVM のバックエンドとリンカを用いて実行ファイルを生成する．
(4) DF-Salvia 独自のコマンドにより，(2) で生成した解析結果を基に DF Tables を作成し，DF Tables を実行ファイルに組み込む．

5.2 DF-Salvia Linux

DF-Salvia Linux は，資源にアクセスするシステムコールと DFTS を対象にし，DF Tables を基にデータフローの追跡・制御を行う．対象のシステムコールが発生された場合，DF-Salvia Linux は，スタックバックトレースを行い，システムコールが発行されたユーザ空間内の戻りアドレスを取得する．これにより，戻りアドレスを用いて DF Tables の Def-Use Table から命令に使用される変数を特定し，データフローの追跡処理を行う．

また，資源にアクセスするシステムコールは，資源の利用目的に応じて以下の処理を行う．ファイルからデータを読み出すシステムコールの場合，データを読み出すファイルに関連付けられたポリシを変数に関連付ける．さらに，データを資源に出力するシステムコールの場合，変数に関連付けられたポリシに従って出力の可否を判定する．

DFTS には，以下の 7 種類のシステムコールが存在する．これらのシステムコールが行う 4.3 節で示した動的なデータフローの追跡について述べる．なお，システムコール名の横に付けた番号は，4.3 節中の番号に対応する．

- policy_delete (5)
  このシステムコールは，変数に定数を格納する処理の前に挿入される．このシステムコールを受け取った DF-Salvia Linux は，変数に関連付けられたポリシを外す．
- policy_prop_assign (1)
  このシステムコールは，代入，データのコピーを行うライブラリ関数の呼出し処理の前に挿入される．このシステムコールを受け取った DF-Salvia Linux は，変数間のポリシの伝播，または別名の記録を行う．
- policy_prop_call (1) (2)
  このシステムコールは，関数の先頭に挿入される．このシステムコールを受け取った DF-Salvia Linux は，関数のローカル変数に対してデータフローの追跡を開始する．さらに，引数によって発生するデータフローを追跡する．
- policy_prop_return (1) (3)
  このシステムコールは，関数のリターン命令の前に挿入される．このシステムコールを受け取った DF-Salvia Linux は，戻り値のデータフローを追跡する．さらに，関数の引数とローカル変数に対するデータフローの追跡を終了する．
- set_offset (4)
  このシステムコールは，配列やポインタのインデックスを変数で指定する命令の前に挿入される．このシステムコールを受け取った DF-Salvia Linux は，通知されたインデックスを Access Table に記録する．
- alloc_vd (2)
  このシステムコールは，メモリの確保を行うライブラリ関数の呼出し，可変長配列による変数の確保する命令の前に挿入される．このシステムコールを受け取った DF-Salvia Linux は，上記の処理によって生成された変数に対してデータフローの追跡を開始する．
- free_vd (3)
  このシステムコールは，free 関数の呼出しの前に挿入される．このシステムコールを受け取った DF-Salvia Linux は，削除される変数に対してデータフローの追跡を終了する．

6. 評価

本章では，データフローを追跡することでデータを識別し，ファイルのポリシに従ってアクセス制御が動作することを検証する．検証には，サンプルコードを用いた動作検証と一般に使用されるアプリケーションを用いた動作検証を行う．

6.1 サンプルコードを用いた動作検証

2 つのサンプルコードを用いて動作検証を行った．検証で用いるサンプルコードは，二つのファイルを入力とし，一つのファイルに出力するプログラムのソースコードである．検証では，全てのコピーを禁止するポリシを関連付けられた機密情報 SECRET と全てのコピーを許可したポリシが関連付けられた公開情報 PUBLIC を用意した．1 つ目

```
 1  char *get_file_data(FILE *in_file) {
 2      char *file_data;
 3
 4      file_data = malloc(MAX);
 5      fgets(file_data, MAX, in_file);
 6      return file_data;
 7  }
 8
 9  int main(void) {
10      int i;
11      char *pointer[2], buf[MAX];
12      FILE *secret, *public, *out;
13
14      secret = fopen(SECRET, "r");
15      public = fopen(PUBLIC, "r");
16      out = fopen(OUT, "w");
17      pointer[0] = get_file_data(secret);
18      pointer[1] = get_file_data(public);
19
20      memcpy(buf, pointer[0], MAX);
21      fputs(buf, out);
22      fputs(pointer[1], out);
23      for (i = 0; i < 2; i++)
24          free(pointer[i]);
25      return 0;
26  }
```

図 4 ポインタ，関数呼出し，動的確保の処理を持つプログラム

の検証には，**図 4** のサンプルコード (以下，サンプルコード 1) を用いる．この検証では，ポインタ，関数呼出し，動的確保 (malloc 関数による領域の確保) に対してデータフローを追跡できることを確認した．2 つ目の検証には，**図 5** のサンプルコード (以下，サンプルコード 2) を用いる．この検証では，構造体のメンバを識別できることと動的な型変換が行われた場合でもデータフローを追跡できることを確認した．

6.1.1 サンプルコード 1

図 4 のサンプルコード 1 の処理の流れについて述べる．

(1) 14–15 行目で，公開情報と機密情報のファイルを開く．
(2) 17 行目で，入力ファイルからデータを読み出すために get_file_data 関数 (1–7 行目) を呼び出す．get_file_data 関数は，引数に指定されたファイルのデータを動的に確保した領域に格納し，その領域のアドレスを戻り値として返す．この関数呼出しにより，get_file_data 関数のローカル変数 file_data と引数 in_file の領域が確保される．さらに，仮引数 in_file に実引数 (main 関数のローカル変数 secret) のアドレスが格納される．
(3) get_file_data 関数内の 4 行目では，malloc 関数により領域 (変数) が確保される．さらに，確保した領域のアドレスがポインタ file_data に格納される．
(4) get_file_data 関数内の 5 行目では，fgets 関数により，ファイル記述子 in_file によって示されるファイル (機密情報) のデータが変数 file_data の参照先に格納される．
(5) get_file_data 関数内の 6 行目では，変数 file_data に格納されたアドレスを main 関数に返す．17 行目において，返されたアドレスは main 関数のローカル変数 pointer[0] に格納される．さらに，関数が終了するため，get_file_data 関数のローカル変数 file_data と引数 in_file の領域が解放される．
(6) 18 行目では，get_file_data 関数が呼び出されており，(2)–(5) と同様の処理が行われる．これにより，公開情報のデータが格納された領域のアドレスが main 関数のローカル変数 pointer[1] に格納される．
(7) 20 行目の memcpy 関数は，変数 pointer[0] の参照先を変数 buf にコピーする．
(8) 21–22 行目では，変数 buf と変数 pointer[1] の参照先をファイル out にデータを書き出されている．
(9) 23–24 行目では，配列 pointer の参照先変数がそれぞれ free 関数により削除されている．このとき，配列 pointer のインデックスが変数 i によって指定されている．

上記の処理に対して実行時に DF-Salvia が行うデータフローの追跡処理について述べる．

(2) では，変数が確保されているため，DF-Salvia は，get_file_data 関数のローカル変数 file_data と引数 in_file に対してデータフローの追跡を開始する．さらに，引数のデータフローを追跡するために，仮引数 in_file と実引数 (main 関数のローカル変数 secret) の別名を記録する．

(3) において，DF-Salvia は，malloc 関数により確保された変数のデータフローの追跡を開始する．さらに，変数 file_data と生成された変数の別名を記録する．

(4) において，DF-Salvia は，変数 file_data の参照先である malloc 関数によって生成された変数にファイルのポリシを関連付ける．

(5) において，戻り値を受け取る変数 (main 関数のローカル変数 pointer[0]) と malloc 関数によって生成された変数の別名を記録する．

(6) では，(5) と同様に，戻り値を受け取る変数 (main 関数のローカル変数 pointer[1]) と malloc 関数によって生成された変数の別名を記録する．

(7) では，データフローを追跡するために，変数 pointer[0] の参照先変数に関連付けられたポリシを変数 buf に伝播させる．

(8) において，DF-Salvia は，fputs 関数によるデータの出力を制御する．21 行目では，OS は，変数 buf に関連付けられたポリシ (ファイル SECRET のポリシ) に従ってアクセス制御を行う．また，22 行目では，pointer[1] の参照先変数に関連付けられたポリシ (ファイル PUBLIC のポリシ) に従ってアクセス制御を行う．

(9) において，OS は，インデックスの値を受け取り，使用される配列 pointer の要素を特定し，その要素の参照先

```
1  struct separate {
2      int member1;
3      int member2;
4  };
5
6  int main(void)
7  {
8      char buf_secret[MAX], buf_public[MAX];
9      FILE *secret, *public, *out;
10     char buf[8];
11     struct separate *sep;
12
13     secret = fopen(SECRET, "r");
14     public = fopen(PUBLIC, "r");
15     out = fopen(OUTPUT, "w");
16
17     sep = (struct separate *)buf;
18     fgets(buf_secret, MAX, secret);
19     sep->member1 = atoi(buf_secret);
20     fgets(buf_public, MAX, public);
21     sep->member2 = atoi(buf_public);
22
23     fprintf(out, "%d\n", sep->member1);
24     fprintf(out, "%d\n", sep->member2);
25     return 0;
26 }
```

図 5 動的な型変換の処理を持つプログラム

変数に対するデータフローの追跡を終了する．

このプログラムに対して検証を行った結果，上記の通りにデータフローの追跡が行われたことにより，機密情報のポリシによるアクセス制御が行われ機密情報の書込みが禁止された．また，出力ファイルには公開情報のみが記録されたため，DF-Salvia によるアクセス制御が確認できた．

6.1.2 サンプルコード 2

図 5 のサンプルコード 2 は，文字列型の変数を int 型の要素を 2 つ含む構造体に変換し，それぞれの要素に異なるファイルのデータを格納し，出力ファイルに書き込む．

型変換の処理を行う 17–21 行目の処理について詳しく述べる．

(1) 17 行目では，配列 buf のアドレスをポインタ sep に格納している．このとき，二つの変数の型が異なるため，配列 buf の型 (char 型) をポインタ sep の型 (struct separate 型) に変換することが明示されている．

(2) 18–19 行目では，数値に変換した機密情報をポインタ sep の参照先の要素 member1 (buf[0–3] の領域) に格納している．

(3) 20–21 行目では，数値に変換した公開情報をポインタ sep の参照先の要素 member2 (buf[4–7] の領域) に格納している．

上記の処理に対して実行時に DF-Salvia が行うデータフローの追跡処理について述べる．

(1) において，DF-Salvia は，ポインタ sep と配列 buf の別名を記録する．

```
1  $ ./tftp 192.168.128.1
2  tftp> put /home/salvia0/demo/secret.txt secret.txt
3  tftp: sendto: Permission denied
4  tftp> put /home/salvia0/demo/public.txt public.txt
5  Sent 38 bytes in 0.0 seconds
```

図 6 tftp コマンドを用いた検証の結果

(2) では，配列 buf を struct separate 型に分割してデータを格納するため，DF-Salvia は struct separate 型に従って，配列 buf を要素 member1(buf[0–3] の領域) と要素 member2(buf[4–7] の領域) に分割する．さらに，配列 buf の要素 member1 に機密情報のポリシを関連付ける．

(3) において，DF-Salvia は，(2) で分割された配列 buf の要素 member2 に公開情報のポリシを関連付ける．

サンプルコード 2 に対して検証を行った結果，上記の通りにデータフローの追跡が行われたことにより，サンプルコード 1 の検証と同様に公開情報のみが出力ファイルに書き込まれた．以上から，DF-Salvia によるアクセス制御が確認できた．

6.2 アプリケーションを用いた動作検証

DF-Salvia がファイルのコピーや送信を行うアプリケーションに対してデータフローを追跡することで，ポリシに従って出力の可否を判定し，情報漏洩を防止できることを確認した．そのため，TFTP を使用したファイルの送信を行う tftp [6] を用いて動作検証を行う．

tftp により，機密情報 (secret.txt) と公開情報 (public.txt) を LAN 上にある別の PC に送信し，機密情報の送信が禁止され，公開情報だけが送信されることを確認した．検証の結果を**図 6** に示す．2，3 行目で，機密情報を送信しようとして，3 行目のエラーメッセージから送信が禁止されたことがわかる．また，4，5 行目から，公開情報が送信できることが確認できる．以上のことから，tftp コマンドに対してポリシに従ったアクセス制御を行うことが確認できた．

7. 関連研究

データフローを追跡する技術は，静的に追跡するものと動的に追跡するものに分けられる．静的な手法では，プログラムの実行前にデータフローを解析し，動的な手法では，プログラムの実行時にデータフローを解析する．

7.1 静的手法

静的手法は，主に型に基づいた情報流解析が用いられる．型に基づいた情報流解析は，プログラムを解析しタイプシステムに基づいてデータフローがセキュリティラベルに違反しないことを保証する [7]．このような型に基づいた情報流解析は，既存の言語を拡張することにより実現される [8,9]．プログラマは，セキュリティラベルに基づき拡張言語でプログラムを記述することにより，プログラマの

意図したデータフローを静的時に保証できる．

しかし，セキュリティラベルは，プログラマによって決定されるため，情報漏洩を保護できるかはプログラマに依存する．それに対し，DF-Salvia は，コンパイラによって自動的にデータフローを解析し，プログラマに依存せずに情報漏洩を防止する．

7.2 動的手法

動的手法は，動的バイナリ変換 [10]，特殊なハードウェア拡張 [11]，ソースコードの変換 [12] によって実現されている．

動的バイナリ変換 [10] や特殊なハードウェア [11] を用いてデータフローを追跡する手法は，実行する命令に対してデータフロー追跡処理を挿入する．また，ソースコードの変換 [12] を用いたデータフローの追跡手法は，C 言語のソースコードに対してデータフローを追跡するための処理を挿入する．挿入された処理は，アドレスによってデータを識別する．

それらに対し，DF-Salvia では，ソースコードからデータフローを解析し，変数と型に基づいてデータを識別する．そのため，プログラミング言語レベルでデータフローを追跡することができ，より明確に意味のあるデータを識別できる．

8. おわりに

本論文では，DF-Salvia のためのコンパイラと OS の連携によりデータフローを追跡する手法について述べた．この手法は，コンパイラにてソースコードを解析し，データフロー情報の生成と OS と連携するためのシステムコールの挿入を行うことにより，OS にてデータフローを追跡する．これにより，プログラム言語のレベルでデータフロー追跡することができ，意味のあるデータを識別できる．また，この手法によりデータフローが追跡できることをサンプルコードと tftp を用いて確認した．

今後は，関数ポインタによるライブラリ関数の呼出しや実行ファイルとライブラリ間で共有される大域変数について対応を行う．

参考文献

[1] NPO ネットワークセキュリティ協会：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 USENIX Security Symposium*, pp. 29–42 (2001).

[3] Harada, T., Horie, T. and Tanaka., K.: Task Oriented Management Obviates Your Onus on Linux, *Proceedings of Linux Conference*, pp. 1–8 (2004).

[4] Ida, S., Kashiyama, T., Takimoto, E., Saito, S., Cooper, E. W. and Mouri, K.: Design and Implementation of DF-Salvia which Provides Mandatory Access Control based on Data Flow, *Proceedings of the International Multi-Conference of Engineers*, pp. 14–16 (2012).

[5] Lattner, C. and Adve, V.: LLVM: a compilation framework for lifelong program analysis transformation, *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 75–86 (2004).

[6] Holland, D. A.: NetKit, `http://freecode.com/projects/netkit` (2015).

[7] Sabelfeld, A. and Myers, A. C.: Language-based informationflow security, *IEEE Journal on Selected Areas in Communications*, Vol. 21, No. 1, pp. 5–19 (2003).

[8] 古瀬　淳，米澤明憲：VITC: 対攻撃耐性コード生成コンパイラ，コンピュータソフトウェア， Vol. 25, No. 1, pp. 180–185 (2008).

[9] Myers, A. C., Zheng, L., snd S. Chong, S. Z. and Nystrom, N.: Jif: Java information flow. Software release, `http://www.cs.cornell.edu/jif` (2015).

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

[11] de Amorim, A. A., Collins, N., DeHon, A., Demange, D., Hritcu, C., Pichardie, D., Pierce, B. C., Pollack, R. and Tolmac, A.: A Verified Information-Flow Architecture, *Proccedings of the 41st ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pp. 165–178 (2013).

[12] Lam, L. C. and cker Chiueh, T.: A General Dynamic Information Flow Tracking Framework for Security Applications, *Proccedings of the 22nd Annual Computer Security Applications Conference*, pp. 463–472 (2006).

© 2015 Information Processing Society of Japan