情報処理学会論文誌 Vol.56 No.12 2313–2323 (Dec. 2015)

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,†1] 奥野 航平[1,†2] 大月 勇人[1] 瀧本 栄二[1] 毛利 公一[1,a)]

受付日 2015年3月9日, 採録日 2015年9月2日

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,†1] Kohei Okuno[1,†2] Yuto Otsuki[1] Eiji Takimoto[1] Koichi Mouri[1,a)]

Received: March 9, 2015, Accepted: September 2, 2015

Abstract: There are many information leakage incidents that are caused by human error. To prevent them, we have developed DF-Salvia that controls output processing of data based on a policy set to each file by a user. DF-Salvia tracks data flow inside a process by cooperation between compiler and OS. In this paper, we describe a method to track the data flow. In DF-Salvia, the compiler analyzes source code, creates data flow information, and inserts additional code into source code to track the data flow dynamically. The OS tracks the data flow in run-time according to them. In the results of applying our method to applications, we have confirmed that DF-Salvia can track the data flow and prevent information leakage.

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あげられる．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 1 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あげられる．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

[1] 立命館大学
Ritsumeikan University, Kusatsu, Shiga 525–8577, Japan
[†1] 現在，株式会社東芝
Presently with TOSHIBA CORPORATION
[†2] 現在，株式会社インターネットイニシアティブ
Presently with Internet Initiative Japan, Inc.
[a)] mouri@cs.ritsumei.ac.jp

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，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．

以上の背景から，正当なアクセス権限を持つユーザによる人為的ミスをヒューマンファクタとし，それに起因す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うえ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，5 章でデータフロー追跡手法の実装，6 章で評価について述べる．また，7 章で関連研究について述べ，8 章でまとめる．

2. DF-Salvia

2.1 機能の概要

DF-Salvia は，情報漏洩の約 80%を占める人為的ミスによる情報漏洩を防止することを目的としたセキュアシステムである．具体的には，正規のアクセス権限を持つユーザが起こす誤操作や管理ミスなどにより，機密情報が直接漏洩することを防ぐ．制御フローを通じて発生する暗黙的なデータフローについては対象としない．また，外部の攻撃者による攻撃や正規ユーザによる意図的な情報漏洩は対象としない．

DF-Salvia によるアクセス制御の概要を**図 1** に示す．DF-Salvia では，ファイルを単位としてポリシを設定することができる．ポリシには，ファイルに格納されたデータが，どの出力先に出力されてもよいのかよくないのかといった出力先の制限を記述できる．DF-Salvia は，ファイルに設定されているポリシに従ってアクセス制御を行うことで，ユーザが機密情報を運用する際に起こす人為的ミスによる情報漏洩を防止する．たとえば，図 1 では，機密情報にすべ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．ほかにも，「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ができる．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．

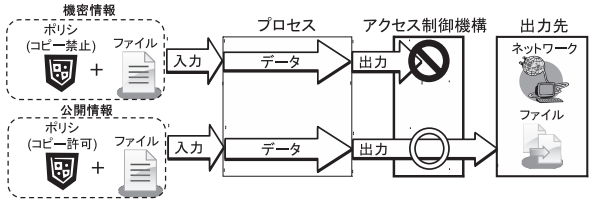

図 1 DF-Salvia によるアクセス制御
Fig. 1 Access control by DF-Salvia.

このようなアクセス制御を実現するためには，データの出力時に，そのデータの源となるファイルを特定する必要がある．そのため，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．DF-Salvia は，コンパイラと OS の連携により，システム全体に対して上記のアクセス制御の機能を提供する．これにより，システム管理者は，OS のインストールとアプリケーションのコンパイルのみで，DF-Salvia を導入可能である．すなわち，機密情報を取り扱うための特別なアプリケーションの開発や既存のアプリケーションの変更をする必要はない．

2.2 保護ポリシ

情報は一般的にファイルとして記憶装置に保管される．ファイルには，内容や目的に応じてデータが格納されるため，ファイルごとに保護ポリシを設定することは，そのファイルの利用目的に適合した保護を可能にする．これらのファイルを源として発生するデータフローを，そのファイルに関連付いた保護ポリシに従って制御することで，情報の利用目的に応じたアクセス制御が実現できる．保護ポリシに記述可能な主な項目は以下のとおりである．

Read 権限 ファイルに格納されたデータの利用可否を設定できる．拒否に設定した場合，プロセスに対するデータの入力を拒否する．

Send Local 権限 プロセスの外部でかつ計算機の内部へデータ伝播（読み込んだファイル以外のファイルへの

書き込み，別プロセスへのプロセス間通信）の可否を設定できる．拒否に設定した場合，ファイルのコピーやプログラム間での情報共有を防止できる．

Send Remote 権限 計算機の外部，すなわちネットワークを経由して別の計算機へのデータ伝播の可否を設定できる．拒否に設定した場合，ネットワークを経由したデータの送信は拒否される．

ポリシの記述の詳細については文献 [5] を参照されたい．

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（ファイルのデータを格納する点，入力点）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．これによって，どのデータがどこに出力されるか（プロセスの外部に出力する点，出力点）を識別する．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) 入力点の fgets 関数で変数 buf1 にファイルのデータが書き込まれる．
(2) コピーを発生させる strcpy 関数で変数 buf1 のデータが buf2 にコピーされる．
(3) 出力点の fputs 関数で変数 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 へも関連付ける．
(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．

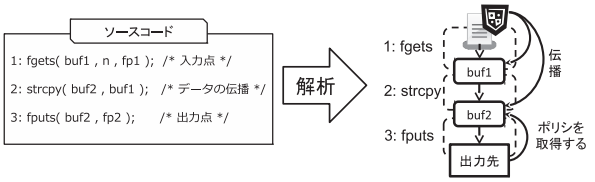

図 2 データフロー情報
Fig. 2 Information of data flow.

3.2 型に基づくデータフロー追跡の粒度

前節で述べたポリシの関連付けをするためには，変数の型に基づいた変数の識別ができなければならない．C 言語では，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．これについては，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．なお，ポリシの異なる複数のデータが 1 つの char 型配列に格納された場合，ポリシのうち 1 つでも出力を禁止するものがあれば出力を拒否する．

3.3 ポインタにおける参照先の記録

ポインタは，アドレスを保持し，そのアドレスに格納されたデータにアクセスする手段を提供する．したがって，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，単純に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，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 キャストによる領域の再分割

C 言語では，キャストによる変数の型変換によって，変数のメモリ領域が分割され，それぞれ異なるデータを格納できる．そのため，型変換によって分割された変数の領域それぞれに格納されるデータを識別する必要がある．本論文では，変数の型変換による領域の分割を変数の分割と表記する．

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. DF-Salvia の構成

4.1 概要

2 章で述べた機能を，3 章で述べた粒度で解析してデータフローを追跡するために，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（**図 3** 参照）．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）の呼び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 DF Tables

DF-Salvia ツールチェインのコンパイラは，変数を読み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．これにより，各命令文を実行したときに，変数から変数への伝播を特定できる．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み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，読み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．関数呼び出しにより生成される変数を特定するために用いる．

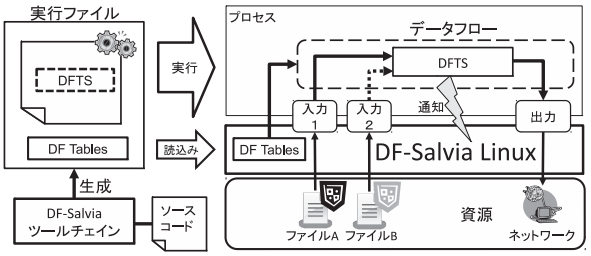

図 3 DF-Salvia の構成
Fig. 3 DF-Salvia architecture.

4.3 DFTS

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 の呼び出しを挿入する．

(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) 配列などにおけるアクセス先の決定
(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る際に DFTS が発行され，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，ポリシが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．

(2) は，ローカル変数の確保，ライブラリ関数（malloc 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際に DFTS が発行され，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，ローカル変数を持つ関数 func が呼び出された場合，DFTS を起点にして，OS は Function Frame Table から関数 func が生成するローカル変数を特定する．さらに，特定したローカル変数に対して，データフローを追跡できるよう，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．また，malloc(4); の呼び出しが存在した場合，DFTS を起点としてサイズ 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．

(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，OS は，Function Frame Table から関数 func が呼び出し時に生成したローカル変数を特定し，そのデータフローの追跡を終了する．

(4) は，配列などにおいて，インデックス変数を用いてアクセスするような場合の処理である．このような場合ではインデックス変数の値が実行時まで分からないため，DFTS を通じて OS へインデックス変数の値を通知し，アクセスする要素を特定可能にする．たとえば a = b[i]; であれば，DFTS により変数 i とその値を使用する命令が OS に通知される．OS は，変数 i をアクセス情報を書き換えるために使用し，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する．

(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，ポリシは削除しなければならない．よって，DFTS を契機として，OS は，Def-Use Table から定数値を格納する変数を特定し，当該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 [6] を用いて実装し，DF-Salvia Linux は，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 の呼び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 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行された場合，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．read などの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，write などの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．なお，アクセス制御によって入出力が拒否された場合，システムコールは EACCES（Permission denied）を返し，ユーザプログラムに入出力の失敗を通知する．なお，mmap システムコールを用いた入出力については現在対応していないが，ページフォルトハンドラを活用したメモリアクセス単位で制御する方法や，マップされた領域を動的に確保されたメモリ領域と同様に扱い，ファイルへ反映されるときにまとめて制御をかける方法がある．

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び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び出し，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 関数の呼び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，2 つのファイルを入力とし，1 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，すべてのコピーを禁止するポリシ（Read 許可，Send Local と Send Remote 不許可）を関連付けられた機密情報「SECRET」とすべてのコピーを許可したポリシ（Read，Send Local，Send Remote すべて許可）が関連付けられた公開情報「PUBLIC」を用意した．

1 つ目の検証には，**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，関数呼び出し，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．

```
 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 サンプルコード 1
Fig. 4 Sample code 1.

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．この関数呼び出しにより，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．

```
 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 サンプルコード 2
Fig. 5 Sample code 2.

(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] の参照先がファイル out に書き出されている．

(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，上記のとお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き込みが禁止された．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ものである．

動的な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，2 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

```
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
Fig. 6 Result of executing tftp command.

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．

(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，上記のとお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．TFTP を使用してファイルを送信する tftp [7] を用いて動作検証を行う．

検証には，6.1 節の検証と同様に，すべてのコピーを禁止するポリシ（Read 許可，Send Local と Send Remote 不許可）を関連付けられた機密情報（secret.txt）とすべてのコピーを許可したポリシ（Read，Send Local，Send Remote すべて許可）が関連付けられた公開情報（public.txt）の 2 つファイルを用いる．tftp により，機密情報と公開情報を 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分かる．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えることが確認できた．

また，典型的な人為的ミスによる情報漏洩として，機密情報のコピー先や送信先を誤ったケースや，GUI アプリケーションを誤操作したケースなどがあげられる．そこで，これらの状況を想定し，下記のアプリケーションについても検証を行った．本検証も tftp の検証と同様のポリシを付与したファイルを用いて行い，いずれもポリシに従ったアクセス制御が行えることを確認した．

- scp コマンドによるファイルの送信
- mailx コマンドによるメールへのテキストファイルの添付
- GUI で動作するファイルマネージャ Thunar を用いた USB メモリへのドラッグ&ドロップによるファイルのコピー
- GUI で動作するメーラ Sylpheed を用いたメールへのテキストファイルの添付

以上から，実用的なアプリケーションでも，DF-Salvia のアクセス制御により，情報漏洩の防止が可能であるといえる．

6.3 性能評価

DF-Salvia のデータフロー追跡による実行時間の増加を計測するために，I/O バウンドな処理と CPU バウンドな処理について計測した．計測用のプログラムには，それぞれ cp コマンドと gzip コマンドを用いる．また，ファイルには，すべてのコピーを許可したポリシ（Read，Send Local，Send Remote すべて許可）を付与する．

実行時間の計測は，同じ PC にインストールされた DF-Salvia Linux と通常の Linux（Normal Linux）上で行う．なお，両者の Linux カーネルのバージョンは，3.9.9 である．計測は，Intel Core 2 Duo 3.00 GHz，メモリ 4 GB を搭載した環境で行った．計測プログラムのコンパイルは，DF-Salvia Linux では DF-Salvia ツールチェインを用いて，Normal Linux では Clang を用いる．また，コンパイラの最適化は無効化し，プログラムは ramfs 上で実行させる．

6.3.1 I/O バウンドな処理

I/O 処理の比率ごとにデータフロー追跡処理のオーバヘッドの変化を確認するために，cp コマンドでコピーするファイルサイズを 1 MB，10 MB，100 MB に変更して実行時間を計測した．その結果，実行時間の増加率は，ファイルのサイズが 1 MB の場合に 3.49 倍となり，10 MB の場合に 1.4 倍，100 MB の場合に 1.001 倍となった．この結果から，I/O 処理の比率の大きい場合にオーバヘッドの割合が小さくなったことが分かる．これは，データフローの追跡処理は CPU バウンドであるため，I/O の処理が増えたとしてもデータフローの追跡処理が発生しなかったことに起因する．そのため，I/O 処理の多いファイル操作を行うプログラムに関しては，オーバヘッドを少なく DF-Salvia を適用可能であると考えられる．

6.3.2 CPU バウンドな処理

gzip コマンドで 1 KB，10 KB，100 KB のファイルをそれぞれ圧縮し，実行時間を計測した．圧縮するファイルは，`/dev/urandom` を用いてランダムな文字列を書き込んで作成したものを使用する．その結果，実行時間の増加は，ファイルサイズが 1 KB の場合に 382 倍となり，10 KB の場合

表 1 各種テーブルのサイズ（バイト数）
Table 1 Size of tables (byte count).

プログラム名	Def-Use & Access	Variable	Function Frame	Type	合計バイト数
sample1	468	72	72	24	636
sample2	540	60	48	60	708
tftp	29,844	4,404	1,008	492	35,748

に 3,183 倍，100 KB の場合に 17,310 倍となった．6.3.1 項の I/O バウンドな処理と比較して，オーバヘッドが大きくなった．

実行時間が増加した大きな要因は，DFTS の呼び出しとその内部で行われるデータフロー追跡にある．そのため，実行時間を改善するためには，コンパイラによる解析精度を向上させ，挿入する DFTS の数を減らすことで発行回数を抑える必要がある．また，データフロー追跡処理を内部の処理で分類し，それぞれの実行時間を計測したところ，型検査と要素の特定処理がオーバヘッドの大きな要因となっていることが分かった．したがって，型変換と要素の特定処理について処理時間を改善する方法について検討する必要がある．

6.4 DF Tables のメモリ使用量

DF-Salvia は，4.2 節で述べた DF Tables の情報を基にプログラムの変数やデータフローを管理する．これらの 5 つのテーブルはプログラム実行時にメモリ上に読み込まれるため，DF-Salvia はテーブルサイズ分のメモリを消費する．6.1 節で述べたサンプルコード 1 および 2 と，6.2 節で述べた tftp について，コンパイル時に作成される各種テーブルの大きさを計測した．結果を**表 1** に示す．作成されたテーブルは，サンプルコード 1，2 で 700 バイト程度，tftp で 35 KB 程度のサイズである．すなわち，実行時には上記のテーブルサイズ分のメモリを消費する．

さらに，DF-Salvia は，これらのテーブルを基に，変数やデータフローを管理するデータ構造を作成する．特に，実行時に実際にデータが格納される変数の最大数がメモリ消費量に影響を与える．変数は，スタティック変数・グローバル変数（静的に生成される変数），ヒープ領域に確保される変数，スタック上に確保されるローカル変数に大別される．以下に，これらの変数の生成に対する DF-Salvia における実行時のメモリ消費量を予測する方法について述べる．

スタティック変数・グローバル変数に対しては，プロセスの起動時に変数管理用のデータ構造が作成される．そのため，スタティック変数・グローバル変数の数 × 変数管理用のデータ構造のサイズ（36 バイト）分のメモリを消費する．ヒープ領域に格納される変数に対しては，動的メモリ確保の管理データ構造（16 バイト）とその動的メモリ確保によって確保した領域をサイズとする変数管理用データ構造が生成される．そのため，プログラムの動的メモリ確保によるメモリ消費量が予測できれば，ヒープ領域に生成される変数が予測できるため，DF-Salvia の実行時に動的確保によるメモリの消費量を予測できると考えられる．スタック上に確保される変数に対しては，関数呼び出しごとにその関数のローカル変数を管理するデータ構造（16 バイト）と，ローカル変数ごとに変数管理用のデータ構造（36 バイト）が確保される．したがって，スタックの最大伸長が分かれば，おおよそのスタックに確保された変数の数が予測できるため，DF-Salvia の実行時にローカル変数によるメモリの消費量を予測できると考えられる．なお，各種管理用構造体は不要となったときに順次解放される．

以上から，コンパイル時に得られるテーブルのサイズ情報と，スタティック変数・グローバル変数のための変数管理用データ構造によるメモリ消費量，動的メモリ確保によるメモリ消費量，スタックの最大伸長に依存して DF-Salvia の実行時に発生するメモリの消費量が増えると考えられる．

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 [8]．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 [9], [10]．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コンパイル時に保証できる．

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，情報漏洩を防止するためには，拡張言語を用いてアプリケーションを開発する必要がある．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できる．

7.2 動的手法

動的手法は，動的バイナリ変換 [11]，特殊なハードウェア拡張 [12]，ソースコードの変換 [13], [14], [15] によって実現されている．なお，本節では，ネットワークから受信したデータなどを汚れ（テイント）と見なし，その伝播を追跡する動的テイント解析も，動的なデータフロー追跡手法の一種として述べる．

動的バイナリ変換 [11] や特殊なハードウェア [12] 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 [13], [14], [15] 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ことができる．

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，コンパイラで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 で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 5 つのアプリケーションを用いて確認した．また，DF-Salvia の処理時間を計測したところ，I/O バウンドなプログラムの処理時間は 1–3.5 倍程度であったが，CPU バウンドなプログラムは最大で 1 万倍を超えるオーバヘッドが発生した．さらに，評価に使用したサンプルプログラムについてテーブルのサイズを計測し，DF-Salvia 環境でプログラムを動作させた場合のメモリ使用量を見積もる方法を述べた．

今後は，関数ポインタによるライブラリ関数の呼び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必要がある．

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，入手先 ⟨http://www.jnsa.org/result/incident/⟩ (2015).

[2] Loscocco, P. and Smalley, S.: Integrating flexible support for security policies into the Linux operating system, *Proc. USENIX Security Symposium*, pp.29–42 (2001).

[3] Harada, T., Horie, T. and Tanaka., K.: Task Oriented Management Obviates Your Onus on Linux, *Proc. 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, *Proc. International MultiConference of Engineers*, pp.14–16 (2012).

[5] 鈴来和久，一柳淑美，毛利公一，大久保英嗣：Privacy-Aware OS Salvia におけるデータアクセス時のコンテキストに基づく適応的データ保護方式，情報処理学会論文誌：コンピューティングシステム，Vol.47, No.SIG3 (ACS 13), pp.1–15 (2006).

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

[7] Holland, D.A.: NetKit, available from ⟨http://freecode.com/projects/netkit⟩ (2015).

[8] 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).

[9] 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).

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

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

[12] 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, *Proc. 41st ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pp.165–178 (2013).

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

[14] Xu, W., Bhatkar, S. and Sekar, R.: Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks, *Proc. 15th USENIX Security Sym.*, pp.121–136 (2006).

[15] Chang, W., Streiff, B. and Lin, C.: Efficient and Extensible Security Enforcement Using Dynamic Data Flow Analysis, *Proc. 15th ACM Conference on Computer and Communications Security, CCS '08*, pp.39–50 (online), DOI: 10.1145/1455770.1455778 (2008).

内匠 真也 （正会員）

1990 年生．2013 年立命館大学情報理工学部情報システム学科卒業，2015 年同大学大学院情報理工学研究科博士前期課程情報理工学専攻修了，同年（株）東芝研究開発センター，現在に至る．システムソフトウェアの研究開発に従事．

奥野 航平

2013 年 3 月立命館大学情報理工学部情報システム学科卒業．2015 年 3 月同大学大学院情報理工学研究科博士前期課程修了．同年 4 月株式会社インターネットイニシアティブ入社，現在に至る．クラウド基盤構築・運用に従事．

大月 勇人 （学生会員）

1988 年生．2011 年立命館大学情報理工学部情報システム学科卒業，2013 年同大学大学院理工学研究科博士前期課程情報理工学専攻修了．同年同大学院情報理工学研究科博士後期課程情報理工学専攻に入学，現在に至る．オペレーティングシステム，仮想化技術，コンピュータセキュリティ等に興味を持つ．

瀧本 栄二 （正会員）

1976 年生．1999 年立命館大学理工学部情報学科卒業，2001 年同大学大学院理工学研究科博士前期課程修了，2005 年同研究科博士後期課程単位取得退学，同年（株）ATR 適応コミュニケーション研究所専任研究員，2010 年立命館大学情報理工学部情報システム学科助手，現在に至る．主にシステムソフトウェア，無線通信に関する研究に従事．博士（工学）．

毛利 公一 （正会員）

1994 年立命館大学理工学部情報工学科卒業，1996 年同大学大学院理工学研究科修士課程情報システム学専攻修了，1999 年同研究科博士課程後期課程総合理工学専攻修了．同年東京農工大学工学部情報コミュニケーション工学科助手，2002 年立命館大学理工学部情報学科講師，2004 年同大学情報理工学部情報システム学科講師，2008 年同准教授，2014 年同教授となり，現在に至る．博士（工学）．オペレーティングシステム，仮想化技術，コンピュータセキュリティ等の研究に従事．電子情報通信学会，ACM，IEEE-CS，USENIX 各会員．