修士論文

# SWIFT :<br>文字列ごとの情報フロー追跡手法

SWIFT: String-Wise Information Flow Tracking

平成20年2月4日提出

指導教員　坂井 修一 教授

東京大学大学院 情報理工学系研究科
電子情報学専攻

勝沼 聡

# 概要

クロスサイト・スクリプティング，SQL インジェクションなどを突く高次のインジェクション・アタックがセキュリティ上の脅威となっている．動的な情報フロー追跡方式（DIFT）は，このようなアタックを統一のアルゴリズムで防ぐ方式として，最近，注目されている．DIFT では，ネットワーク等を介して入力されたデータを，ハードウェアまたはソフトウェアによって情報フロー上で伝播させ，出力時に，その印付けされたデータの検査を行う．ハードウェア・ベースの DIFT は，ソフトウェア・ベースの DIFT と比べて，多くのプログラムに対して適用可能で，また，実行速度のオーバーヘッドも小さい．しかし，伝播精度については十分とは言えず，誤検出や検出漏れが生じる．

本稿で提案する SWIFT（String-Wise Information Flow Tracking）は，ハードウェアで伝播させることで，既存のハードウェア・ベースの DIFT と同様に，上記で述べたような利点がある．さらに，既存手法とは異なり，命令単位ではなく，よりセマンティックな文字列操作の単位で伝播させることで精度を高める．SWIFT では，文字列操作を識別し，文字列から文字列へ伝播させる．

SWIFT を x86 命令エミュレータ Bochs 上に実装し，その伝播精度に関して，既存のハードウェア・ベースの DIFT と比較することで評価を行った．その結果，SWIFT では，既存手法のような誤検出や検出漏れが生じず，高精度であることが示せた．

# 目 次

# 図 目 次

# 表 目 次

# 第1章　はじめに

## 1.1　背景

近年，サーバに対する不正アクセスによる被害が深刻になっている．Web ページやシステムが改ざん，破壊されたり，サーバ内の機密情報が盗まれたりしている．このような不正なアクセスとしては，バッファオーバーフロー・アタックがよく知られているが，最近では，クロスサイト・スクリプティング，SQL インジェクションなどを突く，よりセマンティックなアタックが，特に深刻になっている．このようなバイナリの構造に依らないアタックは，バッファオーバーフロー・アタックなどのメモリ破壊系のアタックと対比して，高次の（high-level）インジェクション・アタックと呼ばれる [7].

高次のインジェクション・アタックを検出する手法として，Perl のテイントモード [2] を発展させた，動的な情報フロー追跡方式（Dynamic Information Flow Tracking，以下では，**DIFT** と略す.） [15] [14] [8] [11] [19] [20] [7] が最近，研究されている．DIFT では，ネットワーク等を介して入力されたデータにテイントと印付けし，そのテイント情報を，プログラムのデータの依存関係に従って伝播させる．そして，出力時に，テイントと印付けされたデータを検査することで，アタックを検出する．なお，この処理を行うのは，プロセッサのようなハードウェアであっても，インタープリタや VM のようなソフトウェアであってもよい．

## 1.2　目的

ハードウェア・ベースの DIFT では，ソフトウェア・ベースの DIFT とは異なり，適用できるプログラムがほとんど限定されず包括的である．また，実行速度のオーバーヘッドも小さい．しかし，伝播の精度は十分とは言えず，誤検出や検出漏れを引き起こしている．本稿では，SWIFT（String-Wise Information Flow Tracking）を提案する．SWIFT では，ハードウェアで伝播を行うことで，既存のハードウェア・ベースの DIFT [7] と同様に，包括的かつ，実行速度のオーバーヘッドも小さい．さらに，既存手法とは異なり，テイント情報を命令単位ではなく，よりセマンティックな文字列操作の単位で伝播させることで高精度な伝播を実現する．

## 1.3 構成

本稿の構成としては，まず，2 章で，高次のインジェクション・アタックについて具体的に説明する．3 章では，DIFT について具体的な手法を挙げつつ，その利点や欠点について論じる．次に，4 章で，本稿で提案する SWIFT の基本的な方針について述べ，5 章で，詳細な動作を説明する．そして，6 章で，SWIFT の伝播精度に関する評価の結果と，その結果に対する考察について述べる．また，7 章で，関連する研究について簡単に説明する．

# 第2章　高次のインジェクション・アタック

## 2.1　概要

メモリ破壊系のアタックとは異なり，バイナリの構造に依存しない，高次のインジェクション・アタックが深刻になっている．図 2.1 は，CVE [5] に報告されたアプリケーションの脆弱性の中で，高次のインジェクション・アタックを引き起こす脆弱性の全報告数に占める割合を示したものであり，年々，増加傾向にあることが分かる．

このような脆弱性があるサーバ・アプリケーションでは，クライアントから入力されたデータが，システム・リソース出力時に，システムに渡されるデータの一部となる．すなわち，図 2.2 のように，入力データはプログラムに文字列として取り込まれた後，複製，結合，変換などの文字列操作で，文字列から文字列に移動し，その移動先の文字列が，ファイル，ネットワークなどへの出力時にシステムによって使われる．高次のインジェクション・アタックでは，このようなプログラムに対し，その入力検査をすり抜けることで，意図していない出力を行わせる．

以下で，具体的なアタックについて，その攻撃先のシステム・リソースの種類ごとに述べる．

## 2.2　ファイル出力時の注入

### 2.2.1　Directory Traversal

Web サーバなどでは，クライアントから入力されたファイル名を，ファイル・アクセス時にそのパスとして使用されることがある．その時に，クライアントからのデータに対し適切な検査を行っていないと，ディレクトリ・トラバーサルが生じる．そして，攻撃者によって，本来アクセスされるべきでないファイルへアクセスされる．

例えば，/www/document というディレクトリの下にあるファイルにクライアントがファイル名を指定してアクセスするという状況を考える．この時，サーバでは，

```
$path = "/www/document/".$name;
```

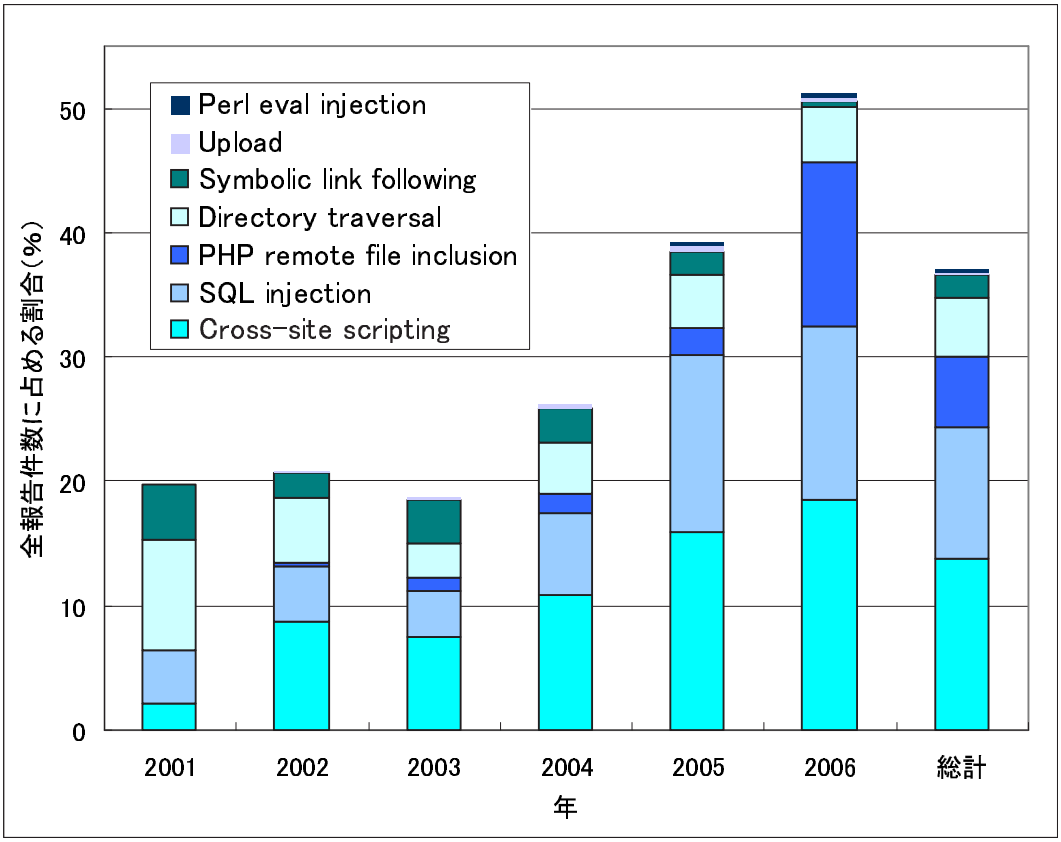



図 2.1: CVE に報告されたアプリケーションの脆弱性

```
fopen($path,...);
```

のように，クライアントからの入力が取り込まれた文字列 name と，ディレクトリ名が結合され，文字列 path が生成される．そして，その path が，関数 fopen でファイル・パスとして用いられる．例えば，

```
/index.html
```

のようなクライアントからの入力が行われたときに，ファイル/www/document/index.html へアクセスされる．このサーバに対して，攻撃者から，

```
/../../etc/passwod
```

というファイル名を指定されると，/www/document 外の/etc/passwod へアクセスされる．

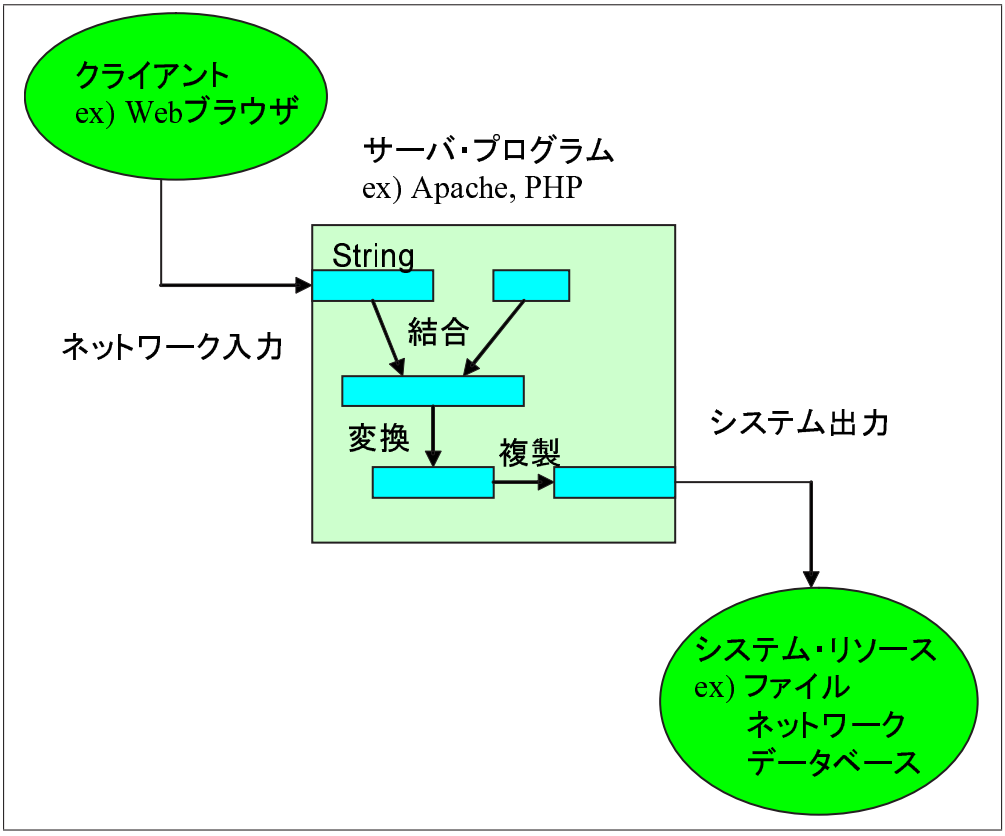



図 2.2: 脆弱性があるサーバ・アプリケーション

### 2.2.2 PHP Remote File Inclusion

PHP のプログラムでは，関数 include や関数 include_once などで，指定されたファイル・パスを読み込んで実行する．したがって，そのファイル・パスをクライアントから指定できると，リモートにあるコードを読み込まされて，実行されうる．

例えば，プログラムに

```
$exec_path = $page . '.php';
include($exec_path);
```

ようなコードがあったとする．このコードは，クライアントからの入力を受け取った文字列 page から，文字列 exec_path が生成され，その文字列を関数 include に渡す．例えば，

```
/index.php?page=archive
```

という入力が行われたときに，archive.php が実行される．このサーバに対して，攻撃者から，

```
/index.php?page=http://www.malicious.code.com/C99.php?
```

という入力が行われたときに，

```
http://www.malicious.code.com/C99.php
```

のような，攻撃者が指定したリモートのプログラムが実行されることになる．

### 2.2.3 Symbolic Link Following

サーバによっては，シンボリック・リンクを用いて，ファイルの処理を行っている．この時，シンボリック・リンクの設定が適切に行われていないと，元のファイルのアクセスが許可されていないにもかかわらず，そのシンボリック・リンクによってアクセスすることを許可されてしまう．攻撃者は，ファイルのリンク名を指定すると，そのリンク名を取り込んだ文字列がシステムコール open 等に渡され，意図しないアクセスが可能になる．

## 2.3 ネットワーク出力時の注入

### 2.3.1 Cross-site Scripting

動的に生成される Web ページに，クライアントからのデータが使われる場合がある．この時に，入力に対して適切にチェックを行っていないと，XSS が生じる．攻撃者は，ユーザに，スクリプトコードが埋め込まれた入力をさせることで，そのユーザのコンピュータ上でスクリプトコードを自動的に実行させ，クッキー等の情報を漏洩させる．

例えば，あるサーバでは，

```
http://market/search.php?goods=house
```

のようなクライアントからの入力が行われたときに，

```
<HTML>
goods does not exist: house
...
</HTML>
```

のような Web ページを返す．この時，サーバ・アプリケーションでは，入力を取り込んだ文字列 goods とシステムで生成された文字列を結合することで Web ページを生成し，その Web ページをネットワークに出力している．今，クライアント側から

```
http://market/search.php?goods=<script>
document.location="http://attacker/
mal.php?cookie="+document.cookie</script>
```

などのスクリプト・コードを含む入力が行われた時に，

```
<HTML>
goods does not exist: <script>...</script>
...
</HTML>
```

のようなWebページが生成され，クライアント側でスクリプトコードが実行されることとなる．

## 2.4 データベース出力時の注入

### 2.4.1 SQL Injection

サーバ・アプリケーションでは，クライアントから受け取ったデータからSQLクエリを生成し，そのSQLクエリをサーバ側のデータベースに送信されることがよく行われている．その時に，クライアントからのデータに対し適切なチェックを行っていないと，攻撃者によってSQLコマンド等を注入され，データベース内のデータが盗み見や改ざんされたりする．

例えば，サーバ・アプリケーションの一部に，

```
$cmd = "SELECT price FROM products WHERE name ‘".$name."’"
```

のようなPHPコードがあったとする．このコードでは，クライアントからの入力を受け取った文字列nameがシステムが生成した文字列と結合され，文字列cmdを生成される．そして，文字列cmdをSQLクエリとしてデータベースに転送する．攻撃者は，

```
tmp’;UPDATE products SET price = 0 WHERE = ’house
```

のようなSQLコマンドを含んだ入力をすることで，

```
SELECT ... WHERE name = ’tmp’;UPDATE ... ’house’
```

のようなSQL文が生成され，これがクエリとしてデータベースに転送されることで，データベースの改ざんが行われる．

## 2.5 その他の出力時の注入

### 2.5.1 Shell Injection

サーバでは，クライアントからのデータをシェルへ転送するコマンドの一部として用いることがある．この時，適切にチェックを行っていないと，攻撃者によって任意の OS コマンドが実行される．例えば，PHP プログラムに，

```
$sh_do = $sh_comd." ".$host;
system($sh_do, ..);
```

のようなコードがあるとする．PHP の関数 system は引数で指定された文字列をシェルに転送する．上記のコードでは，クライアントから入力された文字列 host と，システムで作られた文字列 sh_comd を連結し，文字列 sh_do を作っている．そして，その文字列を関数 system の引数に渡すことで，シェルでそのコマンドを実行させている．このようなサーバに対して，攻撃者は，

```
?host=127.0.0.1;cat%20/etc/passwd
```

という入力を行うことで，シェルに”cat /etc/passwd”というコマンドが渡され，ファイル/etc/passwd が参照される．

### 2.5.2 XPath Injection

サーバによっては，データを保存するために XML データベースを利用し，その XML データベースへのアクセスのために XPath を用いる．そのようなサーバで，クライアントから入力されたデータを用いて，XPath クエリを動的に生成するとき，その入力に対して適切にチェックされていないと，XML データベース内のデータの完全性や機密性が損なわれる．

例えば，プログラムに，

```
XPathExpression expr =
   nav.Compile("string(//user[name/text()='"
   +TextBox1.Text+"' and password/text()='"
   +TextBox2.Text+"'/account/text()"(;
```

のようなコードがあったとする．このコードでは，クライアントから受け取ったデータを格納した文字列 TextBox1.Text，TextBox2.Text から，文字列 expr を生成している．文字列 expr は，この後，XPath クエリとして使われる．このサーバに対して，攻撃者は，

```
NoUser'] | P //user[name/text()='NoUser
```

という入力を行うことで，許可されていないアクセスを行うことができる．

# 第3章　動的な情報フロー追跡

## 3.1　動作

動的な情報フロー追跡（DIFT）は，高次のインジェクション・アタックを統一のアルゴリズムで検出する．DIFT に関して様々な手法が提案されているのが，基本的に，DIFT では，実行しているプログラムのワード（バイト）に対して，テイントか否か識別を行う．まず，ネットワークなどを介して外部からプログラムに入力されたデータに，ソフトウェアでテイントと印付けする．そして，プログラム実行時に，そのテイント情報をデータの依存関係に従って伝播させる．例えば，

```
a = b;       // b から a に伝播
a = b + c;   // b と c から a に OR 伝播
```

のように，データ転送や演算が行われたときに，参照された変数から代入される変数へテイント情報を伝播させる．テイント情報の伝播は，手法によって，ソフトウェアまたは，ハードウェアによって行われる．そして，プログラムの出力時，すなわち，ファイルやデータベース等へのアクセスや，ネットワークとの通信などが行われた時に，表 3.1 のように，テイント情報が付加したデータの検査を行い，条件を満たしたらアタックとして検出する．

## 3.2　課題

DIFT の課題としては，伝播精度，実行速度，包括性が挙げられる．以下で，それぞれについて述べる．

まず，伝播精度について述べる．テイント情報の伝播は，データの依存関係に従って行う．しかし，データの依存関係には，データの転送や演算など，直接的なデータの依存関係がある場合だけでなく，アドレス依存，暗黙的依存と呼ばれる間接的な依存関係が存在する．以下，それぞれについて説明する．

まず，アドレス依存とは，

```
dest = translation_table[index]; //index から dest に依存
```

のような，変換テーブルのインデックスから，変換先のデータへの依存である．アドレス依存は，エンコード，デコード等の文字コード変換や，大文字小文字変換

表 3.1: DIFT の出力検査の例

| 出力先 | 検査項目 | 検出するアタック |
|---|---|---|
| ネットワーク | テイントな<br>スクリプトタグ | Cross-site scripting |
| データベース | テイントな<br>SQL コマンド | SQL injection |
| ファイル | 公開ファイル以外の<br>テイントなパス<br>（システムコール<br>open，execve 等） | PHP remote file inc.<br>Directory traversal<br>Symbolic link follow. |
| シェル | テイントなコマンド | Shell injection |
| XML | テイントなコマンド | XPath injection |

など多くの変換で存在する．特に，ネットワーク・アプリケーションでは，入力データに対するデコードや，出力データに対するエンコードが多く行われる．次に，暗黙的依存とは，

```
if(cond == '+')
 dest = ' ';      // cond から dest に依存
```

のような，分岐条件として用いられる変数から，分岐中に代入される変数への依存である．暗黙的依存も，URL エンコード・デコード等の変換において見られる．このような依存関係は，直接的なデータの依存関係と異なり，テイント情報を全ての依存先に伝播させると誤検出が生じるため対処が難しく， [7] [19] [20] などの既存の DIFT でも問題点として挙げられている．

また，実行速度に関しては，DIFT では，テイント情報の伝播が実行速度を下げる要因となっており，特に，ソフトウェア・ベースの手法で，オーバーヘッドが大変大きい．また，包括的であるか否か，すなわち，どれくらいのプログラムに対して手法を適用できるのかということも DIFT の課題として挙げられる．

## 3.3 既存手法

既存の DIFT として，インタプリタ方式と，ソースコード変換方式，ハードウェア方式について，以下で述べる．

### 3.3.1 インタプリタ方式

インタプリタ方式としては，Perl のテイントモード [2] が知られているが，最近では，高次のインジェクション・アタックの検出を目的とした手法が検討されている．例えば，PHP インタプリタ上に実現したものとして，Pietraszek らの手法 [15]，Nguyen-Tuong らの手法 [14] などがある．また，Java 仮想マシン上に実現したものとしては，Haldar らの手法 [8]，Livshits [11] らの手法などがある．以下で，Nguyen-Tuong らの手法について述べる．

#### Nguyen-Tuong らの手法

Nguyen-Tuong らの手法では，テイント情報を PHP インタプリタによって管理する．PHP インタプリタに入力されたデータに印付けし，そのテイント情報を，PHP のコマンドの単位で伝播させる．そして，PHP インタプリタから出力される時に検査を行う．

まず，PHP インタプリタに，GET，POST，cookie や，セッション変数などで入力されたら，その入力を受け取った文字列に対してテイントと印付けする．そして，文字列間でデータ転送や変換などが行われたら，その文字列操作のソースからデスティネーションの文字列へテイント情報を伝播させる．伝播の仕方は，PHP の文字列操作関数ごとに定義されている．PHP には，書式変換や，文字列変換，抽出，スペースの除去など様々な文字列操作を行う関数があり，それぞれの関数に対して文字列間のデータの依存関係を定義する．そして，そして，インタプリタから出力されるときに検査を行う．例えば，関数 echo でネットワークに出力されるデータや，関数 mysql_query でデータベースへ送られるクエリ，関数 system,exec,fopen,eval などの引数のファイル・パスなどを検査する．

Nguyen-Tuong らの手法では，各関数の操作におけるデータ依存を把握した上で伝播を行うので精度が高い．しかし，関数の操作方法を全て把握している必要があるので，適応可能なプログラムは限られる．

### 3.3.2 ソースコード変換方式

ソースコード変換方式としては，Xu らの手法 [19] [20] が知られている．以下で，Xu らの手法について述べる．

#### Xu らの手法

Xu らの手法では，C ソースコードを静的に変換し，テイント情報の付加，伝播，出力検査を行うコードを埋め込む．テイント情報の管理は，ビット配列を用いている．ビット配列では，メモリアドレスをインデックスとして，メモリデータの

バイト当たりのテイント情報を格納している．テイント情報は，read などの入力を扱う関数を変換し，入力時に，テイント情報をビット配列に代入させる．そして，データ転送や演算時に，その代入元のテイント情報を，代入先に伝播させる．そして，write などの出力を扱う関数を変換し，出力時に，その出力データのテイント情報を検査する．

Xu らの手法では，間接的なデータ依存に対してアドホックな対処を行っている．Xu らの手法では，ポイント変数は常に非テイントとしている．その代わり，配列アクセス時，

```
y = trans[x];     // x から y に伝播
```

のように，配列のインデックスから，代入される変数にテイント情報を伝播させる．このことで，アドレス依存に対処する．しかし，この方法では，上記のコードを，配列アクセスからポインタを介したアクセスに変更した，

```
y = *(trans + x); // x から y に伝播させず
```

のようなコードでは伝播させることができず，十分とは言えない．また，暗黙的依存についても，Xu らの手法では，

```
if(x == E){
  ...
  y = E';
  ...
} // E, E' は定数
  // x から y に伝播
```

のようなコードに対して伝播させ，その他の暗黙的依存のあるデータに対しては伝播を行わず，不十分である．

Xu らの手法では，その他の問題点として，

- ソースコードが必要であり，適応可能なプログラムが限られる．
- ソースコードを変換することで，速度のオーバーヘッドが大きくなる．

などが挙げられる．

### 3.3.3 ハードウェア方式

ハードウェア・ベースの方式としては，当初，[17] [6] など，バッファ・オーバーフロー等の低次のインジェクション・アタックを検出する手法が提案されたが，最近では，Raksha [7] など，高次のインジェクション・アタックも検出する手法が提案されている．以下で，Raksha について述べる．

**Raksha**

Raksha は，図 3.1 のように，メモリ，レジスタ，キャッシュ等の記憶領域に対してワード単位のタグを追加し，そのタグによって，データのテイント情報の記録する．まず，システムコールによって入力データが取り込まれたメモリ領域のタグに，OS 等のソフトウェアで印付けする．そして，データ転送命令や演算命令時に，

```
load R1 = [R2]    //[R2] から R1 に伝播
add R1 = R2 + R3 //R2，R3 から R1 に OR 伝播
```

のように，ソースからデスティネーションオペランドに，ハードウェアでテイント情報の伝播を行い，システムコールで出力された時に，ソフトウェアで検査を行う．

Raksha は，ハードウェアで伝播を行うことで，ソフトウェア・ベースの方式とは異なり，プログラムの動作環境に依存しない．したがって，ほとんど全てのプログラムに対して適用可能であり，包括的である．また，ハードウェアで命令実行と並行してテイント情報が伝播されるので，実行速度のオーバーヘッドも小さい．

しかし，伝播精度に関しては，他の手法と同様に，アドレス依存，暗黙的依存のため不十分である．Raksha は，暗黙的依存に対しては全く伝播を行わない．また，アドレス依存に関しては，

```
load R1 = [R2] // R2 から R1 に伝播
```

のように，アドレス伝播と呼ばれる，ソースアドレスからデスティネーションオペランドに伝播させる機構をハードウェアとして提供している．しかし，誤検出の多さから，標準では使用されていない．

## 3.4 まとめ

既存の DIFT をまとめると，表 3.2 のようになる．インタプリタ方式では，関数単位で伝播させると伝播精度が高くなるが，伝播させることができる関数が限られ包括的ではない．また，より細粒度な命令単位で伝播させると，多くのプログラムで伝播させることができるが，間接的なデータ依存により伝播精度が下がる．また，そのような場合にも，適用可能なプログラムは，各言語で記述されたプログラムに限定されており，バイナリプログラムに対しては対応する手段がない．ソースコード変換方式では，間接的依存に対してアドホックな対処をしており，伝播精度は十分でない．また，手法を適用するためにソースコードが必要であり，適応可能なプログラムは限られる．また，実行速度のオーバーヘッドも大きい．一方，ハードウェア方式は，プログラムの実行環境に依存せず包括的である．また，速度低下が小さい．しかし，伝播精度に関しては，間接的依存のため，やはり十分ではない．

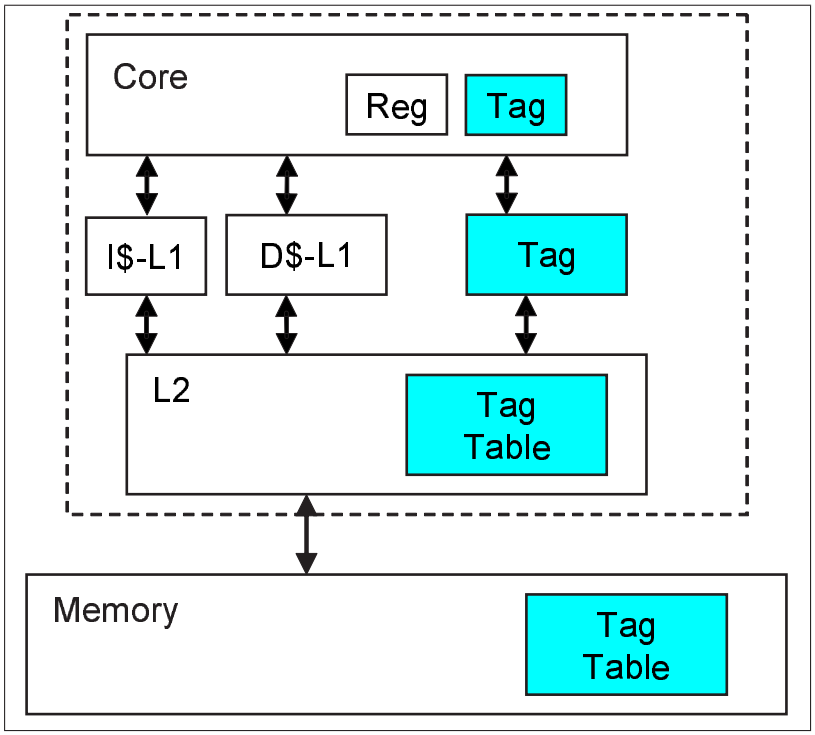



図 3.1: 記憶領域に対するタグ

表 3.2: DIFT

| 方式 | 包括性 | 実行速度 | 伝播精度 |
|---|---|---|---|
| インタープリタ方式 | × | − | △〜○ |
| ソースコード変換方式 | × | × | △ |
| ハードウェア方式 | ○ | ○ | △ |

# 第4章　SWIFTの基本方針

## 4.1　方針

本稿で提案するSWIFTは，テイント情報の伝播方法以外は，Rakshaと同様の方法を取る．SWIFTでは，Rakshaと同様に，ハードウェアでテイント情報を伝播させる．このことで，包括的，かつ，実行速度のオーバーヘッドを小さくする．また，Rakshaとは異なり，命令のソースからデスティネーションに対し伝播を行わない．その代わり，よりセマンティックな文字列操作を識別し，文字列から文字列へデータが移動する時に，その文字列が格納されたメモリ領域のテイント情報を伝播させる．このことで，SWIFTは，命令レベルのデータの依存の仕方に透過的な伝播を行う．つまり，Rakshaで伝播精度を下げる要因になったアドレス依存や暗黙的依存に対しても対応可能にする．

SWIFTでは，まず，インオーダなメモリアクセス（ロード及びストア）のパターンを，動的に取得する．そして，そのパターンに基づき，アクセスされたロード及びストアデータが文字列の要素か否か識別する．また，文字列の識別と同時に，ロードされた文字列からストアされた文字列に移動が行われているかを識別する．そして，移動が行われていると識別されたら，その移動元から移動先にテイント情報を伝播させる．

## 4.2　文字列の識別

一つの文字列の要素は，一般的にメモリの連続領域に格納されており，文字列要素へのメモリアクセスは連続的に行われる．そこで，本手法では，アドレスなどのメモリアクセスのパターンから，連続的に増加または減少するアドレスの列を検出し，そのアドレス列を文字列へのアクセスとみなす．

例えば，表4.1（a）のような，メモリアクセスが行われた時，連続的に増加するロードアドレスの列32〜35を，同じ文字列の要素のロードと識別し（表4.1（b））．また，ストアアドレスの列64〜67を，同じ文字列要素へのストアを識別する（表4.1（c））．

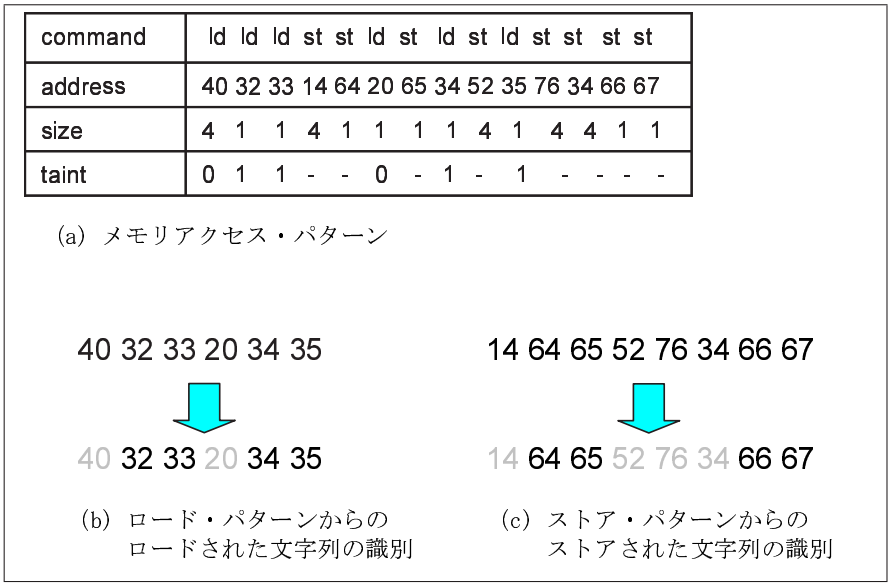



図 4.1: 文字列識別の例

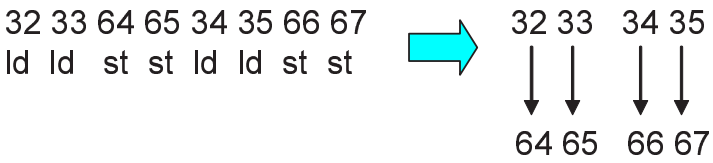
## 4.3 文字列操作の識別

文字列から文字列への移動は，文字列の複製や，抽出，結合のような単純なデータの転送から，文字コード変換や，大文字・小文字変換のような計算を要する変換まで様々であるが，その特徴として，移動元の文字列の各要素から，移動先の文字列の各要素が生成されるという点が挙げられる．すなわち，メモリレベルでは，移動元の文字列要素のロードと，移動先の文字列要素へのストアが繰り返し行われる．そこで，本手法では，ある文字列の要素のロードと，他の文字列の要素へのストアが一定範囲内のサイズで交互に行われたら，その二つの文字列間で移動が行われているとみなす．

例えば，表 4.2（a）では，アドレス 32〜35 から成る文字列とアドレス 64〜67 から成る文字列は，データサイズ 2 バイトずつ，交互にロードとストアが行われていることから，文字列操作が行われているとみなす．そして，アドレス 32，33，34，35 のデータの移動先を，各々，アドレス 64，65，66，67 として，テイント情報を伝播させる（表 4.2（b））．

| command | ld | ld | ld | st | st | ld | st | ld | st | ld | st | st | st | st |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| address | 40 | 32 | 33 | 14 | 64 | 20 | 65 | 34 | 52 | 35 | 76 | 34 | 66 | 67 |
| size | 4 | 1 | 1 | 4 | 1 | 1 | 1 | 1 | 4 | 1 | 4 | 4 | 1 | 1 |
| taint | 0 | 1 | 1 | - | - | 0 | - | 1 | - | 1 | - | - | - | - |

(a) メモリアクセス・パターン




(b) ロード及びストアされた文字列から操作を識別

図 4.2: 文字列操作識別の例

# 第5章　SWIFTの詳細な動作

## 5.1　概要

本章では，前章で述べた方針を実現する方法について詳細に述べる．SWIFT では，メモリやキャッシュに対してバイト当たりのタグを追加し，バイトデータのテイント情報を記憶する．タグの管理に関しては，Raksha と同様の方法を取る．なお，レジスタに関しては，SWIFT ではロードデータからストアデータに伝播を行い，レジスタ上のデータには伝播させないので，レジスタのタグは必要ない．

まず，プログラムに対して外部から入力が行われたら，その入力データが取り込まれたメモリ領域に対して，既存の DIFT と同様に，ソフトウェアでテイントと印付けする．そして，そのテイント情報の伝播を行うが，その方法については既存の DIFT とは異なる．SWIFT では，まず，ロード及びストア命令のインオーダなメモリアクセス・パターンを動的に取得する．そして，ロード命令時に，そのトレース情報をテーブルに記録する．また，ストア命令時には，そのトレース情報を別のテーブルに記憶し，その二つのテーブルの情報に従って，ストア先にテイント情報を伝播させる．そうして，出力時に，既存の DIFT と同様にソフトウェアでテイント情報の検査を行う．

## 5.2　ハードウェア構成

SWIFT は，図 5.1 のように，Load String Table（LST），Store String Table（SST）という二つの CAM のテーブルで構成される．

### 5.2.1　LST

LST は，ロード命令時にその情報を格納するテーブルで，その各エントリには，それぞれ異なる文字列の情報が格納される．LST の項目は次のようになっている．

- *address*（*addr*）：最後にロードされた文字列要素のアドレス
- *size*：最後にロードされた文字列要素のデータサイズ
- *direct*（*D*）：文字列のアクセスの仕方（昇順または降順）

- $ID$：文字列の識別子
- $sum\_size$ （$ss$）：文字列のロードされた要素の合計サイズ
- $taint$：最近ロードされた文字列要素（4 バイト）のテイント情報

### 5.2.2 SST

SST は，ストア命令時にその情報を格納するテーブルである．その各エントリには，LST と同様に，それぞれ異なる文字列の情報が格納される．そして，SST のエントリ内の各 ID のブロックには，その ID に対応づけられた文字列（LST に情報が格納）との文字列操作に関する情報が格納される．SST の項目は次のようになっている．

- $address$ （$addr$）：最後にストアされた文字列要素のアドレス
- $size$：最後にストアされた文字列要素のデータサイズ
- $direct$ （$D$）：文字列のアクセスの仕方（昇順または降順）
- $sum\_load\_size$ （$sls$）：最後にストアされた時の，ID に対応する文字列の合計ロードサイズ
- $curr\_load\_size$ （$cls$）：現在の操作フェーズにおける，ID に対応する文字列要素のロードサイズ
- $curr\_store\_size$ （$css$）：現在の操作フェーズにおける，文字列要素のストアサイズ
- $prev\_size\_ok$ （$ps$）：一つ前の操作フェーズの文字列要素のロード及びストアサイズが条件を満たしているか
- back （b）：戻って伝播させるか否か
- back_taint （b_taint）：戻って伝播させるテイント情報

## 5.3 文字列の識別

### 5.3.1 概要

文字列の識別は，連続的に増加または減少するアドレスの列を検出することで行う．検出方法については，プリフェッチのためのアドレスのストライド予測機 [3]

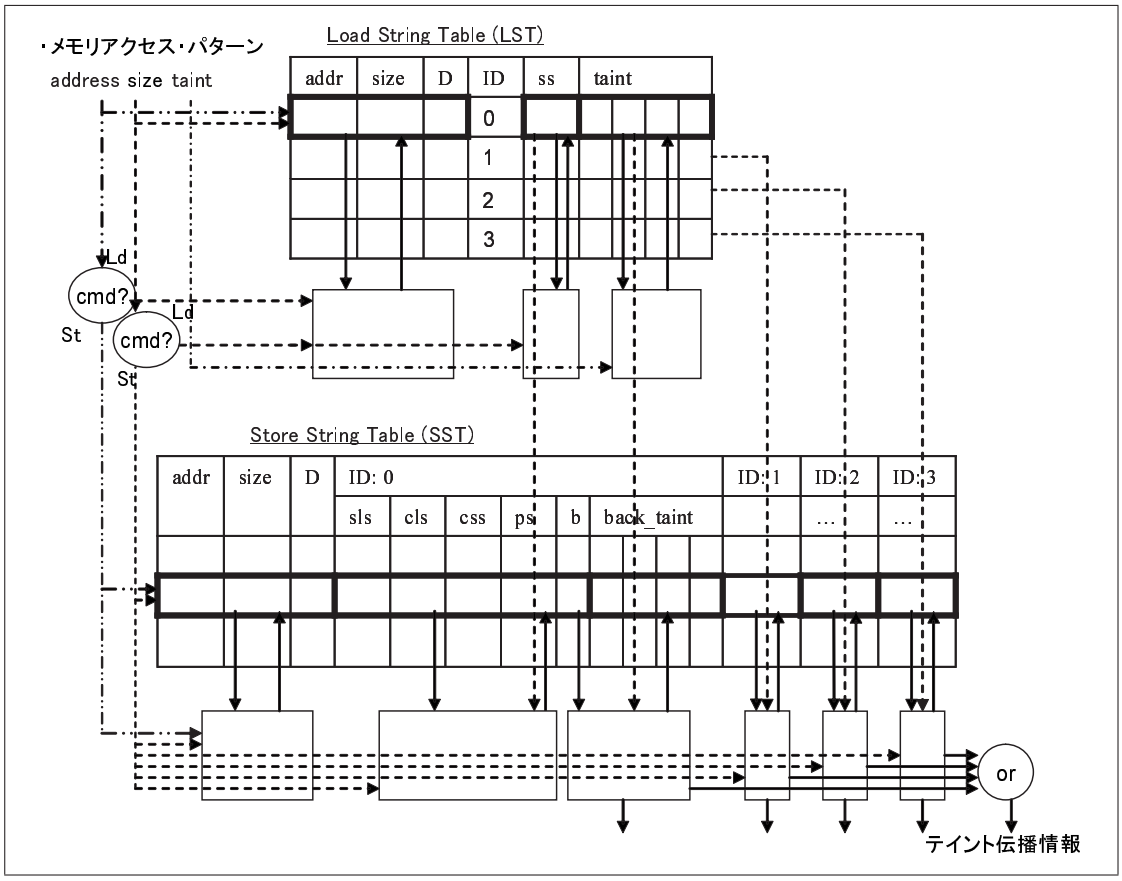



図 5.1: SWIFT のハードウェア構成

に基づいている．ストアされた文字列を識別するために，SST では，各々のエントリに，文字列で最後にアクセスされた要素のアドレスとデータサイズ，その文字列のアクセス順（昇順または降順）を記憶し，ストア命令が行われたときに，そのアドレス，データサイズと各エントリの上記の項目を比較することで，そのストアデータが文字列の次の要素であるかを判定する．ロードされた文字列の識別も同様に，LST を用いて行う．ただし，ロードされた文字列は，テイント情報を持つ要素を含む場合のみ識別される．

### 5.3.2 SST の更新

ストアされた文字列の識別のために，SST の各エントリの項目 *address*，*direct*，*size* を用いる．*address*，*size* は，それぞれ，そのエントリに対応する文字列で最後にアクセスされた要素のアドレス，データサイズを記憶し，*direct* は，その文字列のアクセス順（昇順または降順）を記憶している．

ストア命令時に，その情報に従って，SST を更新する．まず，SST の各エント

リの項目 $address$，$direct$，$size$ を参照し，あるエントリの項目 $address$，$direct$，$size$ と，ストア命令のアドレス $s\_addr$，データサイズ $s\_size$ が，

1. $direct = +, 0 \wedge s\_addr = address + size$
2. $direct = -, 0 \wedge s\_addr + s\_size = address$

のいずれかを満たす場合に，ストアデータを，そのエントリに対応する文字列の次の要素とみなす．そして，そのエントリの $address$，$direct$，$size$ を更新する．$address$ を $s\_addr$，$size$ を $s\_size$ とし，また，$direct$ が 0 の時，(1）では，$direct$ を $+$ に，(2）では，$direct$ を $-$ にする．なお，(1）(2）のいずれかを満たすエントリが存在しない場合，LRU に従って，いずれかのエントリを初期化する．

### 5.3.3 LST の更新

ロードされた文字列の識別には，ストアされた文字列の識別と，同様のアルゴリズムを用いて行う．LST の各エントリの項目 $address$，$direct$，$size$ と，ロード命令のアドレスとデータサイズから，エントリを選択し，更新を行う．ただし，条件を満たすエントリが存在しない場合は，ストア時とは異なり，そのロードデータがテイント情報を持っている場合のみ，いずれかのエントリを初期化する．このようにして，ロードデータに関しては，テイントな要素を含む文字列のみ識別される．

### 5.3.4 動作例

文字列識別の動作を例を挙げて説明する．図 5.2（b）～（e）は，図 5.2（a）のようなメモリアクセスが行われた時の，SST の項目 $address$，$direct$，$size$ の更新の様子である．アドレス 64 へのストア時に，同じ文字列が SST に存在しないので，図 5.2（b）のように，エントリが初期される．そして，アドレス 65 へのストア時に，$65(s\_addr) = 64(addr) + 1(size)$ となるので，アドレス 64 の文字列の次の要素と判定され，図 5.2（c）のように，対応するエントリが更新される．さらに，アドレス 66，67 へのストア時にも，それぞれ，$66 = 65 + 1$，$67 = 66 + 1$ となるので，同じ文字列の次の要素と判定され，図 5.2（d）（e）のように，先程と同様のエントリが更新される．このようにして，ストアされた文字列の識別が行われる．

| command | ld | ld | ld | st | st | ld | st | ld | st | ld | st | st | st | st |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| address | 40 | 32 | 33 | 14 | 64 | 20 | 65 | 34 | 52 | 35 | 76 | 34 | 66 | 67 |
| size | 4 | 1 | 1 | 4 | 1 | 1 | 1 | 1 | 4 | 1 | 4 | 4 | 1 | 1 |
| taint | 0 | 1 | 1 | - | - | 0 | - | 1 | - | 1 | - | - | - | - |

(a) メモリアクセス・パターン

| addr | size | D |
|---|---|---|
| 14 | 4 | 0 |
| 64 | 1 | 0 |
| | | |
| | | |

(b) アドレス64ストア時

| addr | size | D |
|---|---|---|
| 14 | 4 | 0 |
| 65 | 1 | + |
| | | |
| | | |

(c) アドレス65ストア時

| addr | size | D |
|---|---|---|
| 14 | 4 | 0 |
| 66 | 1 | + |
| 52 | 4 | 0 |
| | | |

(d) アドレス66ストア時

| addr | size | D |
|---|---|---|
| 14 | 4 | 0 |
| 67 | 1 | + |
| 52 | 4 | 0 |
| | | |

(e) アドレス67ストア時

図 5.2: 文字列の識別

## 5.4 文字列操作の識別

### 5.4.1 概要

文字列操作の識別は，文字列要素のロードとストアが一定範囲内のサイズで交互に行われるのを検出することで行う．厳密には，次のステップ 1，2 がその順序で 2 回行われると，文字列 A と B は文字列操作が行われているとみなす．

**ステップ 1.** 文字列 A が 1〜4 バイト，ロード．

**ステップ 2.** 文字列 B に 1〜4 バイト，ストア．

ステップ 1，2 の文字列 A，B へのメモリアクセスを合わせて，操作フェーズと呼ぶ．例えば，図 5.4（a）では，アドレス 32，33，64，65 へのアクセスや，アドレス 34，35，66，67 へのアクセスが，それぞれ操作フェーズである．文字列操作を識別するために，SST のエントリの各 ID ブロックに，そのエントリに対応した文字列と，その ID に対応した文字列の関係を記憶しておく．すなわち，現在及び一つ前の操作フェーズのロード及びストアサイズを記録する．そして，その値から文字列操作と識別されたならば，現在及び一つ前の操作フェーズのストア先にテイント情報を伝播させる．

### 5.4.2 SSTの更新

文字列操作を識別するために，SSTのエントリの各IDブロックの$curr_load_size$，$curr\_store\_size$，$prev_size_ok$を用いる．$curr\_load\_size$，$curr_store_size$はそれぞれ，現在の操作フェーズのロードサイズ，ストアサイズを記憶し，$prev_size_ok$は，一つ前の操作フェーズのロード及びストアサイズに関する情報を記憶する．また，$curr\_load\_size$，$curr\_store\_size$，$prev_size_ok$の値を求めるために，LSTの$sum_size$とSSTの$sum_ld_size$を用いる．$sum_size$は，ロードされた文字列要素の合計サイズを記録し，$sum_ld_size$は，最後にストアされた時の，IDに対応する文字列の合計ロードサイズを記録する．

ストア命令時に，ストアされた文字列の識別，すなわち，$address$などの項目の更新と同時に，同じエントリ内の各IDブロックの$curr_load_size$，$curr\_load\_size$，$prev_size_ok$を更新し，その値から文字列操作を識別する．まず，ストア命令時に，$address$を更新したエントリの各IDの$sum_ld_size$と，LSTのその各IDに対応するエントリの$sum_size$（ロード命令時に更新）を比較する．そして，あるIDにおいて，その$sum_ld_size$と$sum_size$の値が等しければ，同じ文字列の一つ前のストアと同じ操作フェーズであるとし，そのIDの$curr_store_size$に，$s_size$を加算する．また，あるIDの$sum_ld_size$と$sum_size$の値が異なれば，同じ文字列の一つ前のストアから操作フェーズが遷移したとみなす．その場合，一つ前の変換フェーズのロードデータ及びストアデータのサイズを記録するために，そのIDの$prev_size_ok$を，

- $prev\_size\_ok = (0 < curr\_ld\_size \leq 4 \wedge curr\_store\_size \leq 4)?1;0$

とする．また，現在の操作フェーズのロードサイズ及び，ストアサイズ等を初期化する．すなわち，

- $curr\_load\_size = sum\_size - sum\_ld\_size$
- $curr\_store\_size = s\_size$
- $sum\_load\_size = sum\_size$

とする．

### 5.4.3 テイント情報の伝播

文字列操作を識別するために，前述のように，ストア命令時に更新された$prev_size_ok$，$curr\_load\_size$，$curr_store_size$の値をチェックする．すなわち，$prev\_size\_ok$，$curr\_load\_size$，$curr_store_size$が，

- $prev_size_ok = 1 \wedge 0 <$curr_ld_size$, curr_st_size \leq 4$

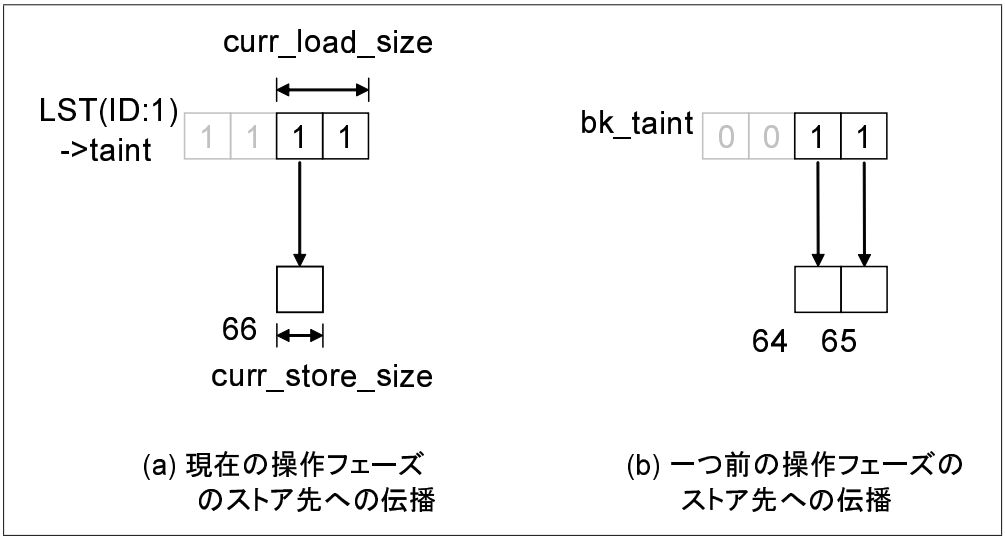



図 5.3: テイント情報の伝播

を満たすのなら，その ID に対応する文字列からストア先の文字列に対して文字列操作が行われているとみなし，現在及び一つ前の操作フェーズのストア先にテイント情報を伝播させる．

現在の操作フェーズのストア先への伝播には，ロードされたデータのテイント情報を記録した，LST の *taint* を参照し，図 5.3（a）のように，テイント情報をストア先に伝播する．また，一つ前の操作フェーズのストア先への伝播には，一つ前の操作フェーズのストア時に，伝播すべきテイント情報を記録した，SST の *bk_taint* を用いる．*bk_taint* を参照し，図 5.3（b）のように，テイント情報を伝播させる．

### 動作例

文字列操作識別の様子を，例を挙げて説明する．図 5.4（b）〜（e）は，図 5.4（a）のようなメモリアクセスが行われた時の，SST における ID:1 のブロックの *curr_load_size*，*curr_store_size*，*prev_size_ok* の更新の様子である．LST の ID が 1 のエントリには，アドレス 32 を基点とした文字列（以下，文字列 I）が対応付けされているとする．アドレス 64 へのストア時，文字列 I では，既にアドレス 32, 33 がロードされているので，図 5.4（a）のように，エントリが初期される．そして，アドレス 65 へのストア時には，文字列 I はアクセスされていないので，操作フェーズは変わらず，図 5.4（b）のように，そのエントリが更新される．さらに，アドレス 66 へのストア時には，文字列 I では，新たに，アドレス 34，35 がロードされたので，操作フェーズが遷移し，図 5.4（c）のようにエントリが更新される．この時，文字列 I から，その文字列へ移動が行われているとみなされる．

| command | ld | ld | ld | st | st | ld | st | ld | st | ld | st | st | st | st |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| address | 40 | 32 | 33 | 14 | 64 | 20 | 65 | 34 | 52 | 35 | 76 | 34 | 66 | 67 |
| size | 4 | 1 | 1 | 4 | 1 | 1 | 1 | 1 | 4 | 1 | 4 | 4 | 1 | 1 |
| taint | 0 | 1 | 1 | - | - | 0 | - | 1 | - | 1 | - | - | - | - |

(a) メモリアクセス・パターン

| addr | ID: 1 | | |
|---|---|---|---|
| | cls | css | ps |
| 14 | 2 | 4 | 0 |
| 64 | 2 | 1 | 0 |
| | | | |
| | | | |

(b) アドレス64ストア時

| addr | ID: 1 | | |
|---|---|---|---|
| | cls | css | ps |
| 14 | 2 | 4 | 0 |
| 65 | 2 | 2 | 0 |
| | | | |
| | | | |

(c) アドレス65ストア時

| addr | ID: 1 | | |
|---|---|---|---|
| | cls | css | ps |
| 14 | 2 | 4 | 0 |
| 66 | 2 | 1 | 1 |
| 52 | 4 | 4 | 0 |
| | | | |

(d) アドレス66ストア時

| addr | ID: 1 | | |
|---|---|---|---|
| | cls | css | ps |
| 14 | 2 | 4 | 0 |
| 67 | 2 | 2 | 1 |
| 52 | 4 | 4 | 0 |
| | | | |

(e) アドレス67ストア時

図 5.4: 文字操作の識別

## 5.5 全体の動作例

LST と SST の全体の更新の様子を，例を挙げて述べる．図 5.5（b）〜（e）は，図 5.5（a）のようなメモリアクセスが行われた時の，LST 及び SST の更新の様子である．アドレス 33 のロード時に，そのロードと同じ文字列を格納したエントリが LST に存在せず，かつ，そのロードはテイント情報を持つので，図 5.5（b1）のように，LRU により選択されたエントリの各項目が初期される．また，アドレス 65 のストア時に，アドレス 64 のエントリと同じ文字列と識別し，図 5.5（c2）のように，そのエントリの各項目を更新する．そして，アドレス 35 のロード時には，図 5.5（d1）のように，アドレス 32 のエントリと同じ文字列と識別され，そのエントリの各項目が更新される．さらに，アドレス 66 のストア時，アドレス 64 のエントリと同じ文字列と識別し，図 5.5（e2）のように，そのエントリの各項目を更新する．また，そのエントリの各項目から，アドレス 64，65 が文字列変換の変換先と判定され，テイント情報が伝播される．また，$first\_taint$ に基づいて，アドレス 64，65 にも伝播される．

(a) メモリアクセス・パターン

| command | ld | ld | ld | st | st | ld | st | ld | st | ld | st | st | st | st |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| address | 40 | 32 | 33 | 14 | 64 | 20 | 65 | 34 | 52 | 35 | 76 | 34 | 66 | 67 |
| size | 4 | 1 | 1 | 4 | 1 | 1 | 1 | 1 | 4 | 1 | 4 | 4 | 1 | 1 |
| taint | 0 | 1 | 1 | - | - | 0 | - | 1 | - | 1 | - | - | - | - |

(b1) アドレス33ロード時のLST

| addr | size | D | ID | ss | taint | | | |
|---|---|---|---|---|---|---|---|---|
| 82 | 1 | + | 0 | 40 | 1 | 1 | 0 | 0 |
| 33 | 1 | + | 1 | 2 | | | 1 | 1 |
| | | | | | | | | |

(b2) アドレス33ロード時のSST

| addr | size | D | ID: 0 | ID: 1 | | | | | | | | ID: 2 | ID: 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | … | sls | cls | css | ps | b | b_taint | | | … | … |
| | | | | | | | | | | | | | |
| | | | | | | | | | | | | | |
| | | | | | | | | | | | | | |

(c1) アドレス65ストア時のLST

| addr | size | D | ID | ss | taint | | | |
|---|---|---|---|---|---|---|---|---|
| 82 | 1 | + | 0 | 40 | 1 | 1 | 0 | 0 |
| 33 | 1 | + | 1 | 2 | | | 1 | 1 |
| | | | | | | | | |

(c2) アドレス65ストア時のSST

| addr | size | D | ID: 0 | ID: 1 | | | | | | | | ID: 2 | ID: 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | … | sls | cls | css | ps | b | b_taint | | | … | … |
| 14 | 4 | 0 | | 2 | 2 | 4 | 0 | 1 | 0 | 0 | 1 | 1 | |
| 65 | 1 | + | | 2 | 2 | 2 | 0 | 1 | 0 | 0 | 1 | 1 | |
| | | | | | | | | | | | | | |

(d1) アドレス35ロード時のLST

| addr | size | D | ID | ss | taint | | | |
|---|---|---|---|---|---|---|---|---|
| 82 | 1 | + | 0 | 40 | 1 | 1 | 0 | 0 |
| 35 | 1 | + | 1 | 4 | 1 | 1 | 1 | 1 |
| | | | | | | | | |

(d2) アドレス35ロード時のSST

| addr | size | D | ID: 0 | ID: 1 | | | | | | | | ID: 2 | ID: 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | … | sls | cls | css | ps | b | b_taint | | | … | … |
| 14 | 4 | 0 | | 2 | 2 | 4 | 0 | 1 | 0 | 0 | 1 | 1 | |
| 65 | 1 | + | | 2 | 2 | 2 | 0 | 1 | 0 | 0 | 1 | 1 | |
| | | | | | | | | | | | | | |

(e1) アドレス66ストア時のLST

| addr | size | D | ID | ss | taint | | | |
|---|---|---|---|---|---|---|---|---|
| 82 | 1 | + | 0 | 40 | 1 | 1 | 0 | 0 |
| 35 | 1 | + | 1 | 4 | 1 | 1 | 1 | 1 |
| | | | | | | | | |

(e2) アドレス66ストア時のSST

| addr | size | D | ID: 0 | ID: 1 | | | | | | | | ID: 2 | ID: 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | … | sls | cls | css | ps | b | b_taint | | | … | … |
| 14 | 4 | 0 | | 2 | 2 | 4 | 0 | 1 | 1 | 1 | 0 | 0 | |
| 66 | 1 | + | | 4 | 2 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | |
| | | | | | | | | | | | | | |

図 5.5: 全体の動作例

# 第6章　SWIFTの評価

## 6.1　評価方法

SWIFTの評価は，x86エミュレータBochs 3.5 [1]上でRed Hat Linux 8.0，apache 2.0，PHP 4.2などを実行させることで行った．Bochsは，x86 CPUの他に，ハードディスク，ネットワーク，CDドライブ，BIOSなど，ほとんど全てのPCシステムをエミュレートしており，Linuxなどの OSを動作させることができる．Bochsを変更し，SWIFTとRakshaを実装した．Rakshaは，ハードウェア・ベースのDIFTで，最も精度が高い手法である．SWIFTは，LST，SSTのエントリ数をそれぞれ4，64とした．また，Rakshaについては，アドレス伝播を行わない方式（以下，Raksha_n）とアドレス伝播を行う方式（以下，Raksha_a）の両方を実装した．Rakshaは，ワード単位のタグを用いているが，公平な評価を行うため，SWIFTに合わせバイト単位のタグを実装した．評価プログラムは，文字列操作の関数とネットワーク・アプリケーションを用い，伝播精度に関して評価を行った．

## 6.2　文字列操作関数

まず，文字列操作関数を用いて，伝播精度の評価を行った．評価方法としては，システムコールread時に，入力データの特定箇所（表 6.2の下線部分）にテイントと印付けし，各関数を実行した後，システムコールwriteで出力されるデータのテイント情報を検査することで行った．

### 6.2.1　評価プログラム

評価プログラムとしては，図 6.1（a）～（j）のようなPHPプログラムを用いた．各々のプログラムは，表 6.1のような文字列操作を行う．(a）～（c）では，抽出，結合，挿入等の文字列間での転送を行い，(f）～（i）では，大文字・小文字変換，URLエンコード・デコード，base64エンコード・デコード等の文字列変換を行っている．また，(d）では，文字列間で移動が起こらない文字列照合を行っている．

```
$str=$_GET["in"];
$dstr =
   substr($str,3,8);
echo $dstr;
```

(a) substr.php

```
$str=$_GET["in"];
$c = "PPP";
$dstr = $c.$str.$c;
 echo $dstr;
```

(b)concat.php

```
$str=$_GET["in"];
$dstr = ereg_replace(
   "(yyy|fghir|zzzz)",
    "PPPPP", $str);
echo $dstr;
```

(c) ereg_replace.php

```
$str=$_GET["in"];
if(ereg(
   "to(nine|night|knight)",
   $str))
   $dstr = "tomorrow";
echo $dstr;
```

(d) ereg.php

```
$str=$_GET["in"];
dstr =
   strtoupper($str);
echo $dstr;
```

(e) strtoupper.php

```
$str=$_GET["in"];
$dstr =
   strtolower($str);
echo $dstr;
```

(f) strtolower.php

```
$str=$_GET["in"];
$dstr =
   urlencode($str);
echo $dstr;
```

(g) urlencode.php

```
$str=$_GET["in"];
$dstr =
   urldecode($str);
echo $dstr;
```

(h) urldecode.php

```
$str=$_GET["in"];
$dstr =
   base64_encode($str);
echo $dstr;
```

(i) base64_encode.php

```
$str=$_GET["in"];
$dstr =
   base64_decode($str);
echo $dstr;
```

(j) base64_decode.php

図 6.1: 文字列操作を行うプログラム

表 6.1: 文字列操作関数による評価 1

| | 関数 | 操作 |
|---|---|---|
| a | substr | 抽出 |
| b | ”.”演算子 | 結合 |
| c | ereg_replace | 挿入 |
| d | ereg | 照合 |
| e | strtoupper | 大文字変換 |
| f | strtolower | 小文字変換 |
| g | urlencode | URL encode |
| h | urldecode | URL decode |
| i | base64_encode | base64 encode |
| j | base64_decode | base64 decode |

### 6.2.2 評価結果

評価の結果，表 6.1 表 6.2 のようになった．表 6.2 では，出力への正しい依存の仕方と，各手法による出力への伝播の様子（ただし,“ − ”は正しい依存の仕方と同じ.）を表している．下線部分が，テイントな箇所である．

（a）〜（c）のようなデータ転送のみを行うプログラムについては，全ての手法で正確に伝播させることができた．また,（d）のような，入力と出力に依存関係がないプログラムでは，全ての手法で伝播しないことが確認できた．一方,（e）〜（j）のような計算を要する変換については，各々の手法で異なる出力結果が得られた．Raksha_n に関しては，多くの検出漏れが生じた．これは，これらのプログラムの変換の過程で，アドレス依存や暗黙的依存が生じているからである．大文字・小文字変換，base64 エンコード・デコードでは全てのデータの変換でアドレス依存が生じ，また，URL エンコードではアルファベット以外のデータの変換でアドレス依存が生じている．また，URL デコードでは,“ + ”の変換で暗黙的依存が生じている．Raksha_a に関しては，全てのプログラムで伝播させることができた．ただし，URL エンコードについては，Raksha_n と同様に，暗黙的依存がある箇所で，部分的に検出漏れが生じた．一方，SWIFT では，全てのプログラムで伝播を確認することができた．しかし,（g）（i）では部分的に伝播できない箇所が存在した．

なお，図 6.2〜図 6.11 は，伝播の様子を表している．図の横軸は，メモリアクセス数を表し，縦軸は，ロードまたはストアアドレスの下位 10 ビットである．青色の点は，テイントなデータのロードを表し，赤色の点は，テイント情報が伝播したストアを表している．

表 6.2: 文字列操作による評価 2

| | 入力 | 正しい依存の仕方 |
|---|---|---|
| a | abcdefrhirklmno | defrhirk |
| b | abcde | PPPabcdePPP |
| c | abcdefrhirklmno | abcdePPPPPklmno |
| d | tonight | tomorrow |
| e | abcdefrhi | ABCDEFRHI |
| f | abcdefrhi | abcdefrhi |
| g | a<b>c; | a%3Cb%3Ec%3B |
| h | a%62+%64e%66r%68i | ab defrhi |
| i | abcdefghi | YWJjZGVmZ2hp |
| j | YWJjZGVmZ2hpamts | abcdefghijkl |

表 6.3: 文字列操作関数による評価結果 1

| | Raksha_n | Raksha_a | SWIFT |
|---|---|---|---|
| a | ○ | ○ | ○ |
| b | ○ | ○ | ○ |
| c | ○ | ○ | ○ |
| d | − | − | − |
| e | × | ○ | ○ |
| f | × | ○ | ○ |
| g | × | ○ | ○ |
| h | ○ | ○ | ○ |
| i | × | ○ | ○ |
| j | × | ○ | ○ |

表 6.4: 文字列操作による評価結果 2

| | Raksha_n による伝播 | Raksha_a による伝播 | SWIFT による伝播 |
|---|---|---|---|
| a | — | — | — |
| b | — | — | — |
| c | — | — | — |
| d | — | — | — |
| e | ABCDEFRHI | — | — |
| f | abcdefrhi | — | — |
| g | a%3Cb%3Ec%3B | a%3Cb%3Ec%3B | a%3Cb%3Ec%3B |
| h | ab defrhi | ab defrhi | — |
| i | YWJjZGVmZ2hp | — | YWJjZGVmZ2hp |
| j | abcdefghijkl | — | — |

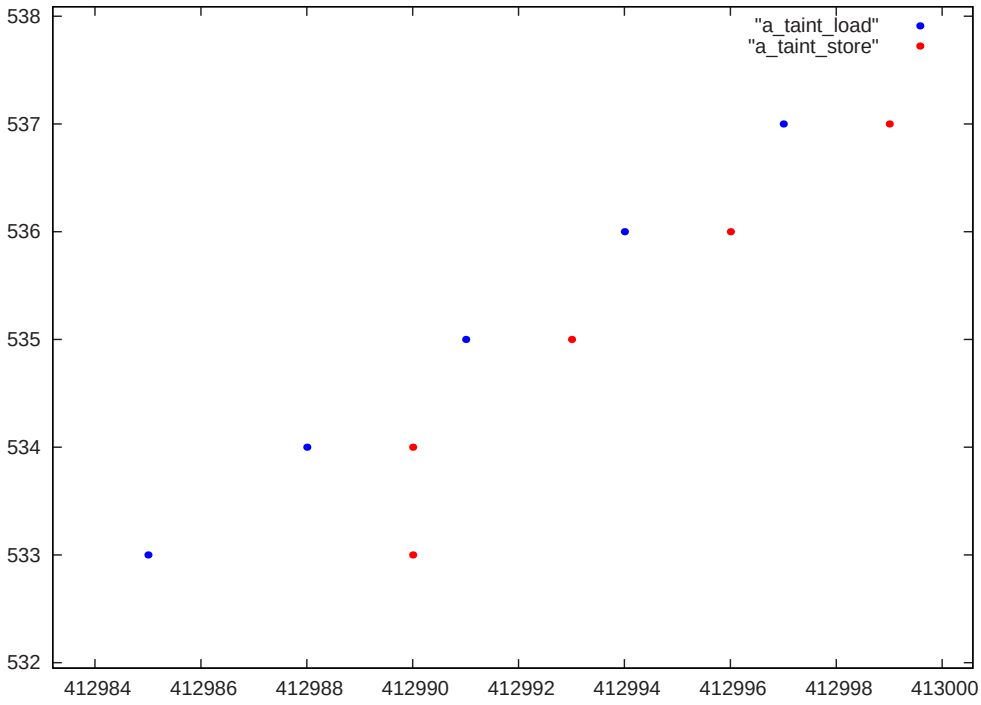



図 6.2: （a）抽出における伝播の様子

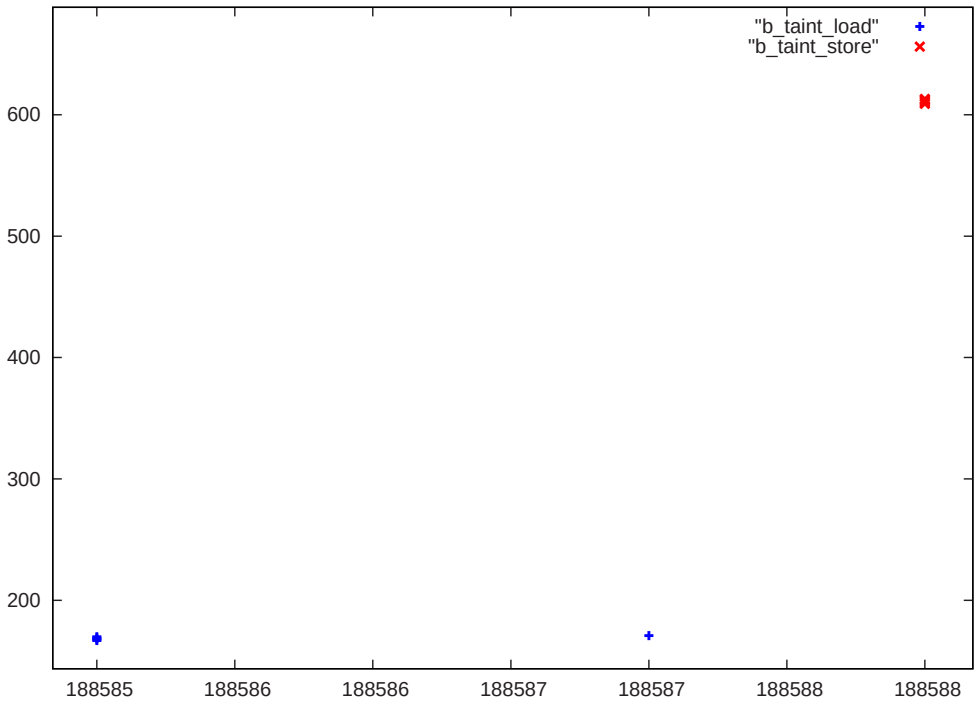



図 6.3: （b）結合における伝播の様子

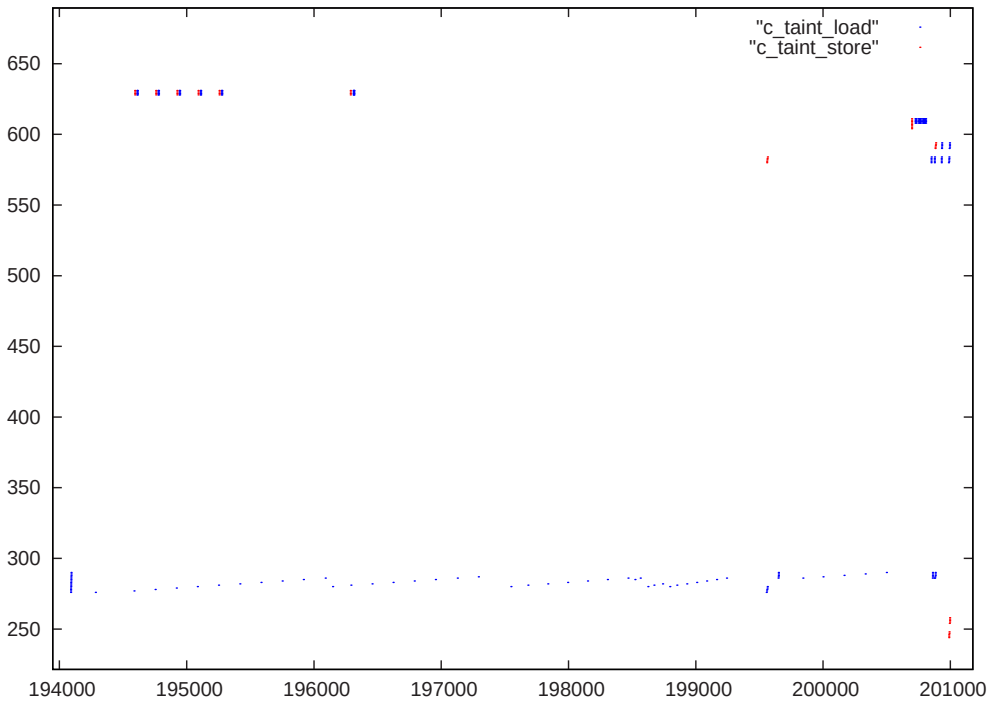



図 6.4: （c）挿入における伝播の様子

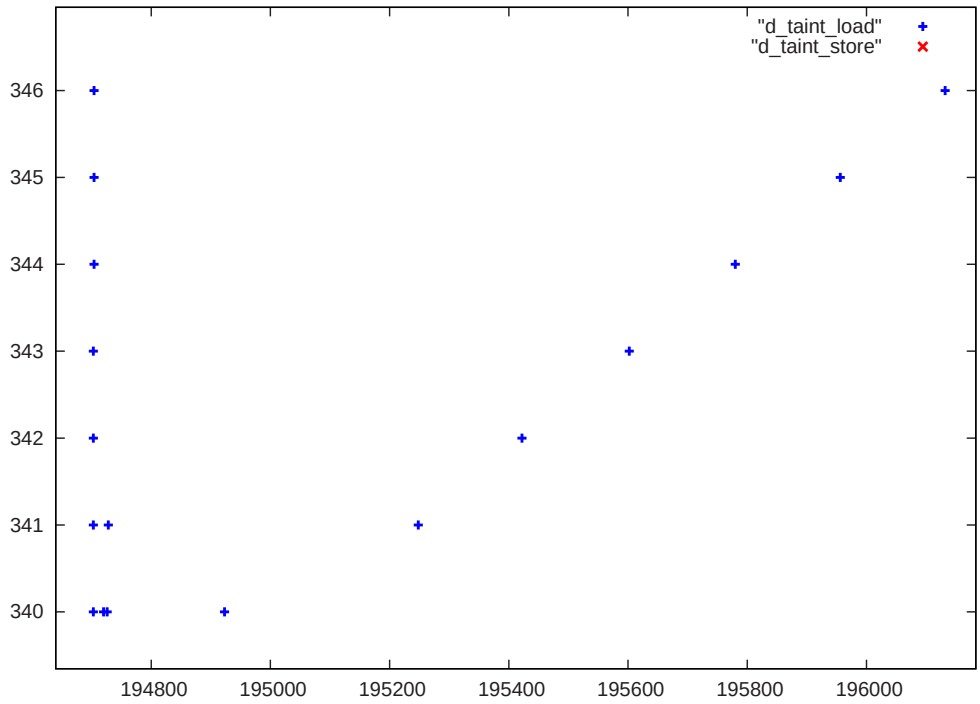



図 6.5: （d）照合における伝播の様子

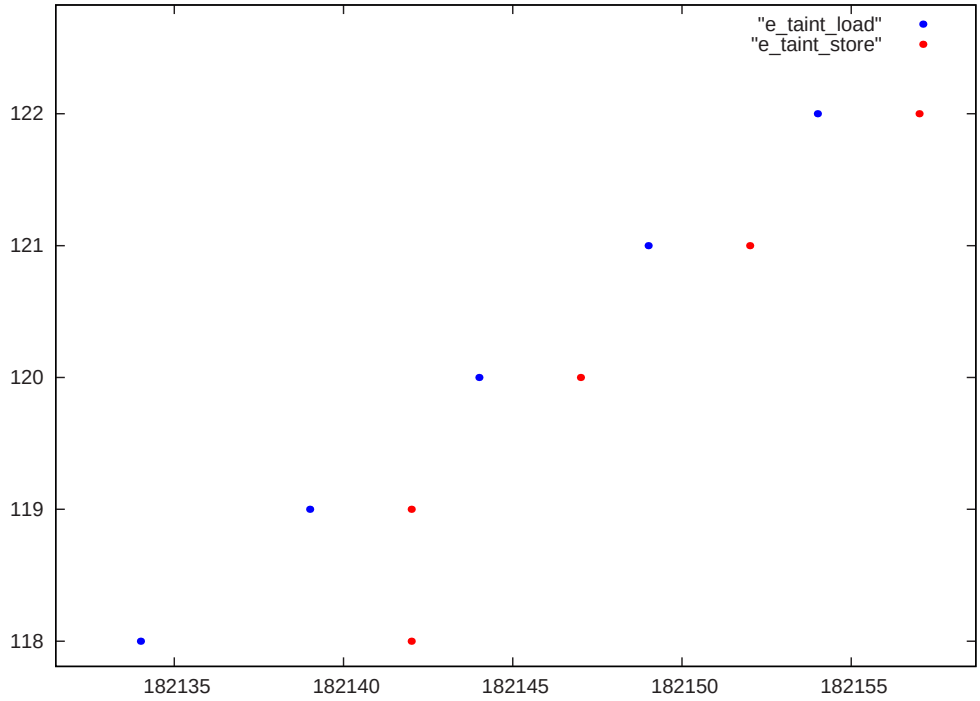



図 6.6: （e）大文字変換における伝播の様子

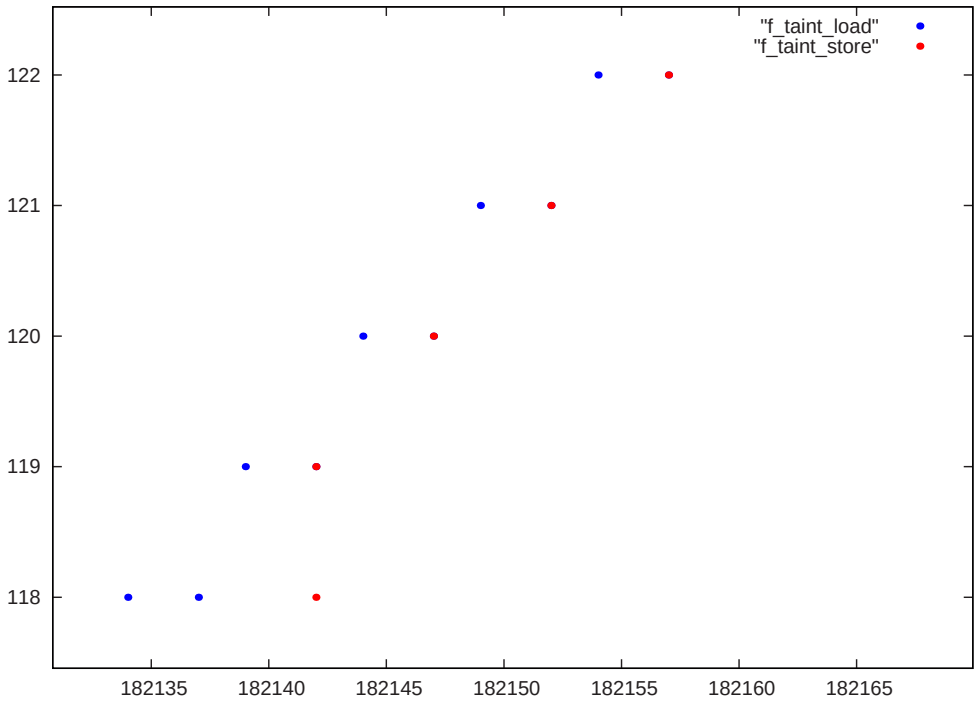



図 6.7: （f）小文字変換における伝播の様子

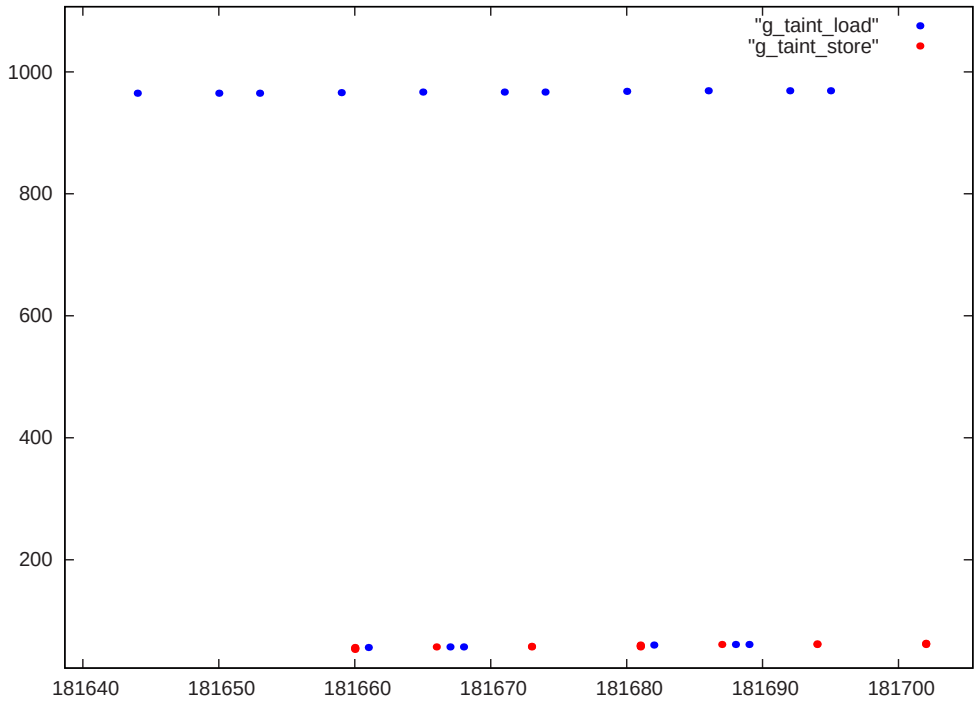



図 6.8: （g）URL エンコードにおける伝播の様子

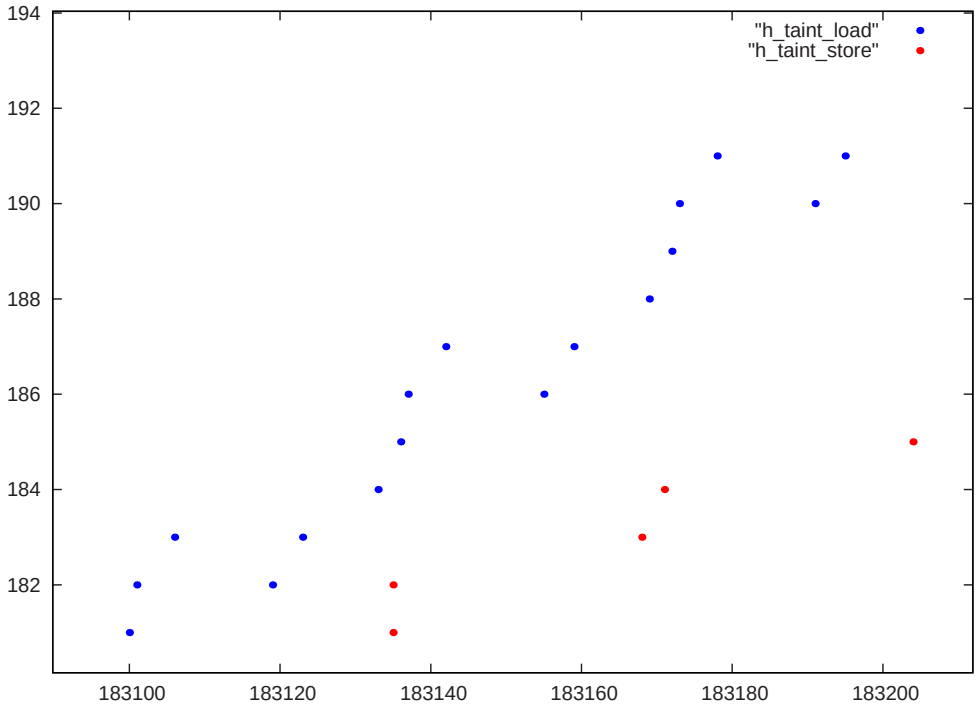



図 6.9: （h）URL デコードにおける伝播の様子

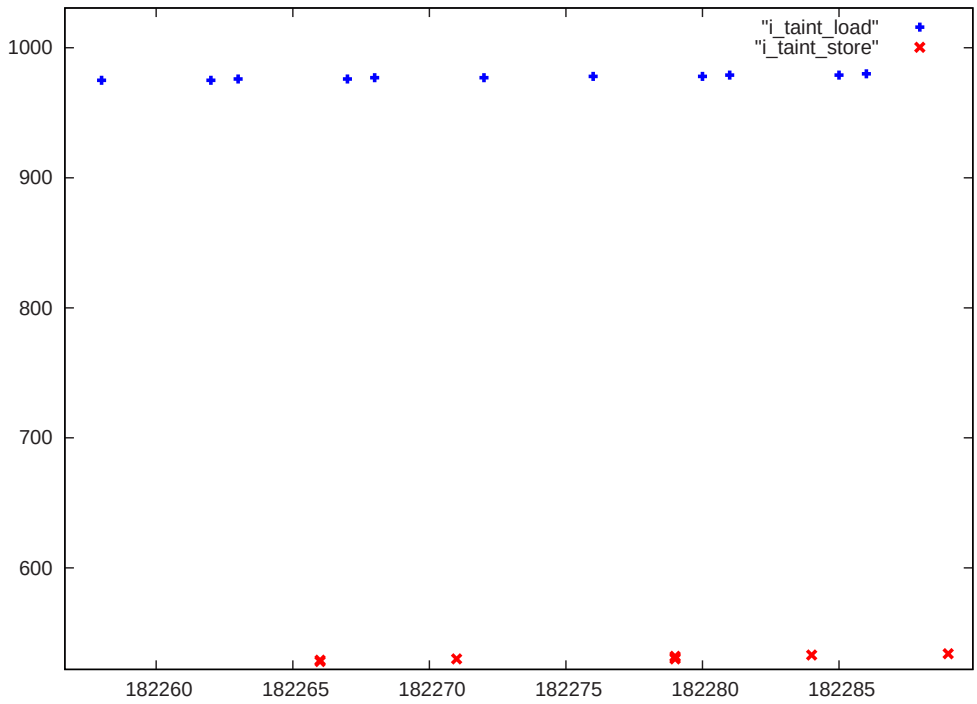



図 6.10: （i）base64 エンコードにおける伝播の様子

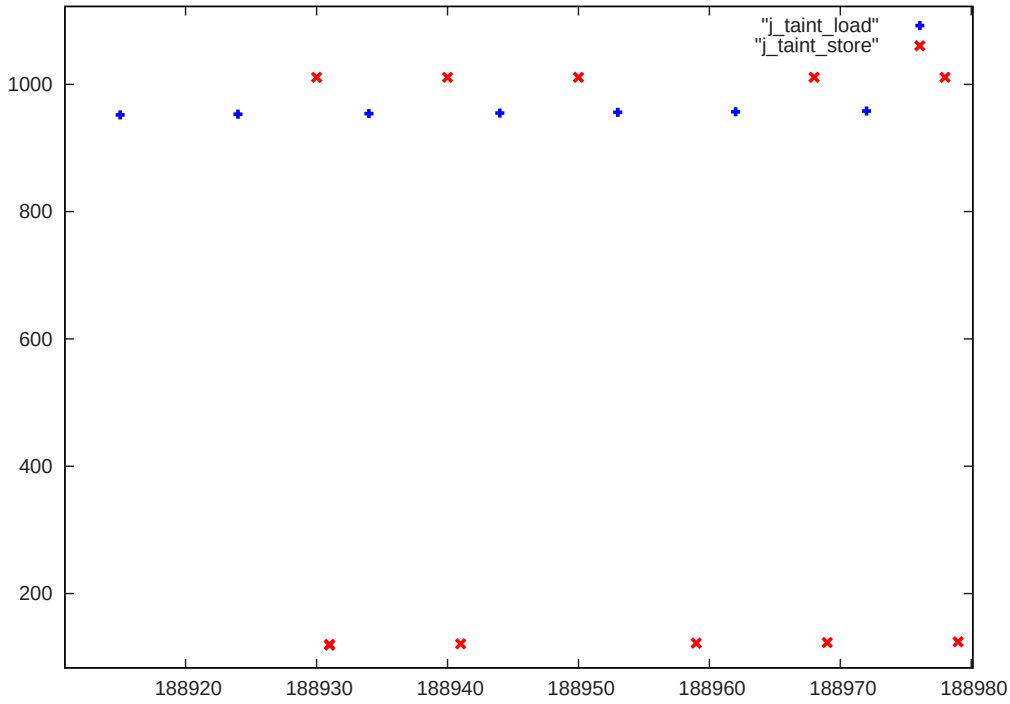



図 6.11: （j）base64 デコードにおける伝播の様子

## 6.3 ネットワーク・アプリケーション

次に，実際のネットワーク・アプリケーションを用いて評価を行った．評価に用いたプログラムは，表 6.5 のようなプログラムであり，各々，高次のインジェクション・アタックを引き起こす脆弱性を持っている．各プログラムに対して，その脆弱性を突くアタックを行い，各手法で検出，誤検出の有無を調べた．なお，出力の検査は，全ての手法で，表 3.1 のように行った．

### 6.3.1 評価プログラム

評価に用いたアプリケーションとその脆弱性について，以下で，それぞれ述べる．

**phpsysinfo 2.3** phpsysinfo では，ファイル index.php において変数 sensor_program が，

```
echo '<center><b>Error:'.$sensor_program.'
is not currently supported</b></center>';
```

のように，ネットワークに出力されるデータの一部として用いられるが，その変数 sensor_program に対して適切にチェックが行われていないため，クロスサイト・

表 6.5: ネットワーク・アプリケーションによる評価

| CVE | Program | Attack type |
|---|---|---|
| 2005-0870 | phpsysinfo 2.3 | Cross site scripting |
| 2003-0486 | phpBB 2.0.5 | SQL injection |
| 2006-0983 | Qwikiwiki 1.4.1 | Directory traversal |
| 2005-2380 | phpSurveyor 0.98 | Cross site scripting |
| 2005-2398 | phpSurveyor 0.98 | SQL injection |
| 2003-1435 | PHPnuke 6.0 | SQL injection |

スクリプティングが生じている．したがって，

```
/index.php?sensor_program=<script>alert('XSS');</script>
```

ように，変数 sensor_program にスクリプトコードを渡して，ファイル index.php にアクセスすると，スクリプトコードを実行させることが可能になる．

**phpBB 2.0.5**　phpBB では，ファイルファイル viewtopic.php で変数 topic_id が，

```
$sql = "SELECT p.post_id
     FROM".POSTS_TABLE."p,".SESSIONS_TABLE."s,".USERS_TABLE."u
     WHERE
     ...
     AND p.topic_id = $topic_id
     ...
     ";
$result = $db->sql_query($sql);
```

のように，SQL クエリの一部として用いられるが，変数 topic_id に対して適切にチェックが行われていないため，SQL インジェクションが生じている．したがって，

```
/viewtopic.php?...&topic_id=-1;DELETE%20%2A%20FROM%20table&...
```

のように，ファイル viewtopic.php アクセス時に，変数 topic_id に SQL コマンドを渡すことで，その注入した SQL コマンドを実行させることが可能になる．

**Qwikiwiki 1.4.1**　Qwikiwiki では，ファイル index.php の変数 page を，ファイル・パスとして用いているが，その変数 page に対して適切にチェックされていないので，ディレクトリ・トラバーサルが生じている．したがって，

```
/index.php?page=../_config.php%00
```

のように，ファイル index.php アクセス時に，変数 page に対して，”..”を含むファイル・パスを渡すことで，アクセスが許可されていないファイルを参照することが可能になる．

**phpSurveyor 0.98**　phpSurveyor では，ファイル browse.php において変数 sid を，

```
$query = "SELECT language FROM {$dbprefix}surveys
      WHERE sid=$sid";
$result = mysql_query($query) or
      die("Error selecting language:
      <br />".$query."<br />".mysql_error());
```

のように，SQL クエリの一部として使い，さらに，その後に，ネットワークへの出力データとして使用されている．しかし，変数 sid に対し適切にチェックが行われていないため，SQL インジェクション及び，クロスサイト・スクリプティングが生じている．したがって

```
/browse.php?sid=1;DELETE%20%2A%20FROM%20table
```

のように，ファイル browse.php アクセス時に，変数 sid に対して SQL コマンドを渡すことで，その SQL コマンドを実行させることができる．また，

```
/browse.php?sid=1<script>alert('XSS')</script>
```

のように，変数 sid にスクリプトコードを渡すことで，そのスクリプトコードを実行させることができる．

**PHPnuke 6.0**　PHPnuke では，ファイル modules.php において変数 days を SQL クエリの一部として用いているが，適切にチェックされていないため，SQL インジェクションが生じている．したがって，

```
/modules.php?...&days=1000;DELETE%20%2A%20FROM%20table&...
```

ように，ファイル modules.php アクセス時に，変数 days に SQL コマンドを渡すことで，その SQL コマンドを実行させることができる．

### 6.3.2　評価結果

評価の結果，表 6.6 のようになった．全ての手法で全てのアタックを検出することができた．また，SWIFT 及び，Raksha_n では誤検出が生じなかった．表 6.7 は，SWIFT において，アタックが検出された出力部分を表している．下線部分が，テイント付けされた箇所である．全てのプログラムで，入力に依存した出力箇所のみテイント付けされ，それ以外の箇所にはテイント付けされなかった．一方，Raksha_a では，プログラムの多くのメモリ領域がテイント付けされ，結果として，全てのプログラムで誤検出が生じた．

表 6.6: ネットワーク・アプリケーションによる評価結果 1

| CVE | Raksha_n | | Raksha_a | | SWIFT | |
|---|---|---|---|---|---|---|
| | 検出 | 誤検出 | 検出 | 誤検出 | 検出 | 誤検出 |
| 2005-0870 | あり | なし | あり | あり | あり | なし |
| 2003-0486 | あり | なし | あり | あり | あり | なし |
| 2006-0983 | あり | なし | あり | あり | あり | なし |
| 2005-2380 | あり | なし | あり | あり | あり | なし |
| 2005-2398 | あり | なし | あり | あり | あり | なし |
| 2003-1435 | あり | なし | あり | あり | あり | なし |

表 6.7: ネットワーク・アプリケーションの評価結果 2

| CVE | 出力 |
|---|---|
| 2005-0870 | : <script>alert('XSS');</script> is not |
| 2003-0486 | id = -1;DELETE * FROM table AND |
| 2006-0983 | data/../_config.php |
| 2005-2380 | sid=1<script>alert('XSS');</script><br /> |
| 2005-2398 | sid=1;DELETE * FROM table |
| 2003-1435 | <= 1000;DELETE * FROM table ORDER |

## 6.4 考察

Raksha_a では，単純なプログラムで，テイント情報を入力データに限定的に付加する場合には，高精度な伝播を行う．しかし，現実的なプログラムでは，多くの誤検出が生じ，現実的な手法とは言えない．このアドレス依存による誤検出は，[19] [7] などの既存の DIFT でも議論されていたことである．また，Raksha_n は，本稿で試したアタックを全て検出することができたものの，多くの典型的な変換で伝播させることができなかった．ネットワーク・アプリケーションにおいて，エンコードやデコードは，それぞれ，入力されたデータや出力されるデータに対して頻繁に行われる．また，入力データを全て小文字に変換するようなアプリケーションも存在する．したがって，このような変換に対応できないのは，本評価で全てのアタックが検出されたとはいえ，安全とは言いがたい．

一方，SWIFT では，本評価で試したアタックを誤検出なく全て検出することができた．また，多くの典型的な変換で伝播させることができた．したがって，SWIFT は，Raksha の両方式よりも伝播精度が高いと言える．しかし，SWIFT にも問題点がある．表 6.1（g）（i）では部分的に伝播できない箇所が存在した．一般に，文

字列操作で移動元と移動先の対応関係を正確に求めるのは難しく，表 6.1（g）（i）のような，移動元と移動先のデータサイズが異なる場合や，文字列操作の最後の操作フェーズでは伝播の精度が下がる．文字列操作の識別精度を向上させることに関しては今後の課題としたい．

# 第7章　関連研究

## 7.1　DIFT

DIFT 方式は，高次のインジェクション・アタックの検出以外に，低次のインジェクション・アタックの検出や，情報漏洩の防止を目的として検討されている．以外で，それぞれの手法について述べる．

### 7.1.1　低次のインジェクション・アタック検出

バッファオーバーフロー・アタックのような，バイナリの構造に依存したアタックは低次のインジェクション・アタックと呼ばれる．低次のアタックの検出手法のアルゴリズムは，検査方法以外は，高次のアタックの検出手法と同様である．入力時にテイント情報を付加し，そのテイント情報を，データの依存に従って伝播させる．そして，検査を，出力時に行うのではなく，各命令実行時に，その命令アドレスやコード，データアドレスがテイントであるか否か調べることで行う．

低次のアタックを検出する手法としては，DIFT [17]，Minos [6]，PTD [4] などのハードウェア方式や，Newsome らの手法 [13]，LIFT [16] などのバイナリコード変換方式が知られている．ハードウェア方式では，Raksha や SWIFT と同様に，ハードウェアによって，テイント情報を管理し，伝播させる．そして，上記で述べたような検査を，また，ハードウェアによって行う．バイナリコード変換方式では，Newsome らの手法では，プログラムをエミュレータ上で動作させ，そのエミュレータでテイント情報の伝播や，検査を行う．また，LIFT は，動的な instrument を用いて，動的にコードを変換し，テイント情報を伝播させたり，検査を行うコードを追加する．

### 7.1.2　情報漏洩防止

情報漏洩を防止する DIFT 方式として，RIFLE [18]，栗田らの手法 [21] が知られている．RIFLE は，静的なバイナリ変換及び，ハードウェアによって伝播させる方式であり，栗田らの手法は，ハードウェアによって伝播させる方式である．これらの手法では，機密情報がプログラムに入力されたら，その入力データに印をつける．そして，情報フロー上で，その入力に依存するデータに伝播させ，出力

時に，その入力に依存するデータが，出力データとして含まれているか否か検査する．

情報漏洩防止手法では，暗黙的依存の解決が課題となっている．なぜなら，アタック対策とは異なり，悪意を持って作成されたプログラムによって，暗黙的依存を利用した情報漏洩が行われるためである．RIFLE は，静的にバイナリ解析を行うことで，また，栗田らの手法はプログラマに強制させることにより，暗黙的依存の解決を行っている．

## 7.2 高次のインジェクション・アタック対策

DIFT 方式以外の，高次のインジェクション・アタック対策として，入力のフィルタリングや脆弱性検査などがある．以下で，それぞれについて述べる．

### 7.2.1 入力のフィルタリング

アタックである可能性が高い入力に対してフィルタリングを行う技術が，アタック対策として，一般的に使われている．入力データの長さを限定する，制御に使われる文字などを含んだ入力を受け取らない，そのような文字を他の文字に置き換えるなどのシグネチャ・ベースの手法が用いられている．また，最近では，入力データがアタックであることをシステマティックに特徴付けるような技術 [?] も検討されている．しかし，入力されたデータを完全に特徴づけるのは難しく，誤検出や検出漏れが生じる．

### 7.2.2 脆弱性検査

プログラム実行前に，あらかじめ，アタックを引き起こす脆弱性を検出する技術が検討されている．静的解析や動的検査などによる様々な手法 [9] [12] が検討されている．これらの手法の問題点としては，正確さが挙げられる．静的解析でプログラムを挙動を正確に把握するのは，ポインタの解析など困難な課題が存在し，一般的には不可能だと言われている．また，動的検査によって，プログラムの全ての実行パターンを検査することも難しい．

## 7.3 アドレスのストライド予測

SWIFT の文字列識別は，プロフェッチのためのアドレスのストライド予測 [3] [10] に基づいている．これらの手法では，ロードアドレスがストライド，すなわち，等間隔になっていることを検出する．検出の仕方は，6 章で述べたようにテーブルに

過去のアドレス値を記録することで行う．そして，ロードアドレスがストライドの関係になっていることが検出されたら，その次の要素をあらかじめキャッシュに転送する．このことでキャッシュ・ミスによる性能低下を抑えている．SWIFT は，ストライド予測と異なり，等間隔なストライドではなく，連続した文字列を検出しているが，そのアルゴリズムには共通点が多く，関連研究として挙げた．

# 第8章　おわりに

## 8.1　まとめ

本稿では，高次のインジェクションアタックを防ぐために，文字列操作の単位でテイント情報を伝播させる方式（SWIFT）を提案した．SWIFT では，文字列操作を識別し，文字列から文字列へのメモリデータの移動にしたがって，テイント情報を伝播させる．評価は，x86 エミュレータ Bochs 上に SWIFT を実装し，既存のハードウェア・ベースの DIFT である Raksha と伝播精度について比較することによって行った．その結果，SWIFT は，Raksha よりも伝播の精度が高いことが示された．

## 8.2　今後の課題

今後の課題としては，伝播精度の向上が挙げられる．文字列操作の識別についてさらに検討することで，前章で述べた検出漏れなどの問題に対処したい．また，速度及びハードウェア容量のオーバーヘッドをついて，ハードウェア・シミュレータなどを用いて，評価，検討を行う必要がある．さらに，命令単位で伝播を行う既存の DIFT との協調に関しても検討したい．

# 参考文献


[1] Bochs: the open source ia-32 emulation project. http://bochs.sourceforge.net.

[2] J. Allen. Perl version 5.8.8 documentation - perlsec. http://perldoc.perl.org/perlsec.pdf, 2006.

[3] J. L. Baer and T. F. Chen. Effective hardware-based data prefetching for high-performance processors. *IEEE Computer Society*, Vol. 44, No. 5, pp. 609–623.

[4] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *International Conference on Dependable Systems and Networks (DSN)*, pp. 378–387, 2005.

[5] S. Christey and R. A. Martin. Vulnerability type distributions in cve. http://cve.mitre.org/docs/vuln-trends/index.html, 2007.

[6] J. Crandall and F. Chong. Minos: Control data attack prevention orthogonal to memory model. In *37th International Symposium on Microarchitecture (MICRO)*, pp. 221–232, 2004.

[7] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A flexible information flow architecture for software security. In *34th International Symposium on Computer Architecture (ISCA)*, pp. 482–493, 2007.

[8] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for java. In *21st Annual Computer Security Applications Conference (ACSAC)*, pp. 303–311, 2005.

[9] Y. W. Huang, F. Yu, C. Hang, C. H. Tsai, D. T. Lee, and S. Y. Kuo. Securing web application code by static analysis and runtime protection. In *13th International Conference on World Wide Web (WWW)*, pp. 40–52, 2004.

[10] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham. Effective stream-based and execution-based data prefetching. In *18th annual International Conference on Supercomputing (ICS)*, pp. 1–11.

[11] B. Livshits, M. Martin, and M. S. Lam. Securifly: Runtime protection and recovery from web application vulnerabilities. In *Technical report, Stanford Univrsity*, 2006.

[12] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *14th USENIX Security Symposium*, pp. 271–286, 2005.

[13] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *20th IFIP International Information Security Conference (SEC)*, 2005.

[14] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *20th IFIP International Information Security Conference (SEC)*, pp. 295–307, 2005.

[15] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pp. 124–145, 2005.

[16] F. Qin, C. Wang, Z. Li, H. S. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *39th International Symposium on Microarchitecture (MICRO)*, pp. 135–148, 2005.

[17] G. Suh, J. Lee, and S. Devadas. Secure program execution via dynamic information flow tracking. In *11th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS-XI)*, pp. 85–96, 2004.

[18] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *37th International Symposium on Microarchitecture (MICRO)*, pp. 243–254, 2004.

[19] W. Xu, S. Bhatkar, and R. Sekar. Practical dynamic taint analysis for countering input validation attacks on web applications. In *Technical Report SECLAB-05-04*, 2005.

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

[21] 栗田弘之, 塩谷亮太, 入江英嗣, 五島正裕, 坂井修一. 動的なインフォメーションフロー制御による情報漏洩防止手法. 情報処理学会報告 2007-ARC-172, pp. 227–232, 2007.

# 発表文献

### 主著論文

- SWIFT: 文字列ごとの情報フロー追跡手法
  勝沼 聡, 塩谷 亮太, 入江 英嗣, 五島 正裕, 坂井 修一
  情報処理学会論文誌コンピューティングシステム（ACS23）, 2008（投稿中）.

- SWIFT: 文字列ごとの情報フロー追跡手法
  勝沼 聡, 塩谷 亮太, 入江 英嗣, 五島 正裕, 坂井 修一
  先進的計算基盤システムシンポジウム 2008（SACSIS2008）, 2008（投稿中）.

- 文字列に着目した情報フロー追跡によるインジェクション攻撃の検出
  勝沼 聡, 塩谷 亮太, 入江 英嗣, 五島 正裕, 坂井 修一
  組込技術とネットワークに関するワークショップ 2008（ETNET2008）, 2008（発表予定）.

- Base Address Recognition with Data Flow Tracking for Injection Attack Detection
  Satoshi Katsunuma, Hiroyuki Kurita, Ryota Shioya, Kazuto Shimizu, Hidetsugu Irie, Masahiro Goshima, and Shuichi Sakai
  IEEE International Symposium on Pacific Rim Dependable Computing (PRDC 2006), pp.165-172, 2006.

- アドレスオフセットに着目したデータフロー追跡による注入攻撃の検出
  勝沼 聡, 栗田 弘之, 塩谷亮太, 清水 一人, 入江 英嗣, 五島 正裕, 坂井 修一
  先進的計算基盤システムシンポジウム 2006（SACSIS2006）, pp.515-524 , 2006.

### 共著論文

- 脆弱性検出のための静的値範囲解析
  樽井 翔, 勝沼 聡, 入江 英嗣, 五島 正裕, 坂井 修一
  電子情報通信学会技術研究報告 CPSY200-22, pp.95-100, 2007.

- 超ディペンダブル・プロセッサアーキテクチャの構想
  入江 英嗣, 荻野 健, 勝沼 聡, 清水 一人, 栗田 弘之, 五島 正裕, 坂井 修一
  電子情報通信学会技術研究報告 CPSY2006-1〜12, pp.49-54, 2006.

# 謝辞

非常に多くの方々から多大なご指導，ご協力，励ましを頂き，本論文を完成させることができました．この場を借りて，感謝の意を表したいと思います．

本研究を進めるにあたり，指導教員である坂井修一教授には，3 年間に渡って多くのご指導，ご助言を頂きました．

また，五島正裕准教授からも，大変多くのご指導を頂きました．ここに深く感謝の意を表します．

入江英嗣博士には研究チームのリーダーとして，様々な形でアドバイスを頂きました．

栗田弘之氏，清水一人氏，塩谷亮太氏をはじめ，ディペンダブル・グループのメンバーの皆様には，ミーティングにおける議論を通して，貴重なご意見を頂きました．

Luong Dinh Hung 氏には，学会発表などに関して，世話していただきました．

研究室の計算機，及びネットワークの構築・管理のために，一林宏憲氏，渡辺憲一氏，杉本健氏には多大なご尽力を賜りました．心より感謝致します．

八木原晴水さん，月村美和さん，内田杏さんには，研究室における設備の導入や各種事務手続きなど，研究室で過ごすための様々なご支援を頂きました．

その他の方々にも，色々な面でお世話になりました．