情報処理学会論文誌　コンピューティングシステム　Vol. 1　No. 2　261–274 (Aug. 2008)

SWIFT：文字列ごとの情報フロー追跡手法

勝沼 聡[†1]　塩谷 亮太[†1]　入江 英嗣[†1,†2]
五島 正裕[†1]　坂井 修一[†1]

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の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．その中で，ハードウェア・ベースの方式は，多くのプログラムに対して適用可能であるが，アタックの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，ハードウェアによる手法であるが，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入力由来のデータを伝播させることで精度を高める．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評価を行った．その結果，SWIFT は，既存のハードウェアベースの DIFT よりも高精度であることが示せた．

SWIFT: String-wise Information Flow Tracking

Satoshi Katsunuma,[†1] Ryota Shioya,[†1]
Hidetsugu Irie,[†1,†2] Masahiro Goshima[†1]
and Shuichi Sakai[†1]

High-level injection attacks, such as SQL injection and cross-site scripting, have a significant effect on computer security. Dynamic Information Flow Tracking (DIFT) methods, which have been proposed nowadays, detect a wide range of these attacks. Hardware-based DIFT works with most programs. However, hardware-based DIFT produces the false-positives and false-negatives. In this paper, we propose a new hardware-based DIFT which is referred to as String-Wise Information Flow Tracking (SWIFT). This technique does not propagate tags by instructions, but by string operations, which leads to high propagation accuracy. We implemented SWIFT on a Pentium-based Bochs emulator and investigated its propagation capability. As the result, it is shown that SWIFT is more accurate than existing hardware-based DIFT.

1. はじめに

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．このような不正なアクセスとしては，バッファオーバフロー・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の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よらないアタックは，バッファオーバフロー・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[5]．**図 1** は，CVE[3] 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[1] 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）[5)–9),11),12)] が最近，研究されている．DIFT では，ネットワークなど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

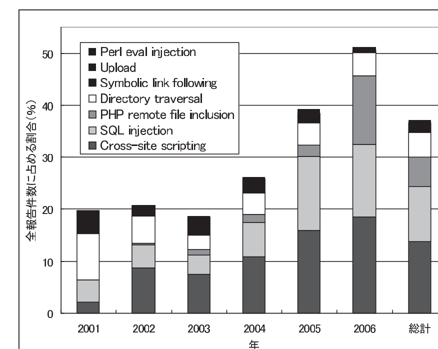


図 1　CVE に報告されたアプリケーションの脆弱性
Fig. 1　Vulnerabilities of applications in CVE.

†1 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
†2 独立行政法人科学技術振興機構
Japan Science and Technology Agency

© 2008 Information Processing Society of Japan

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，インタプリタや VM のようなソフトウェアであってもよい．

ハードウェア・ベースの DIFT では，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，実行速度のオーバヘッドも小さい．しかし，伝播の精度は十分とはいえず，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 [5] と同様に，包括的かつ，実行速度のオーバヘッドも小さい．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．

本稿の構成は，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，DIFT について具体的な手法をあげつつ，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考察について述べる．

2. 高次のインジェクション・アタック

2.1 概　　要

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，ネットワークや，データベースなどのシステム・リソースへの出力の一部として使われる．すなわち，**図 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出力として使われる．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，プログラムが意図していない出力を行わせる．

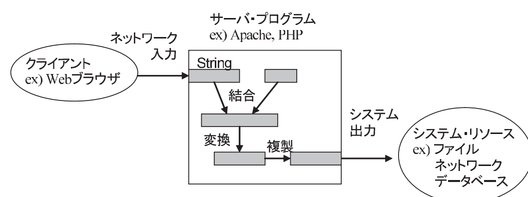


図 2　脆弱性があるサーバ・アプリケーション
Fig. 2　Vulnerable server application.

2.2 SQL インジェクション

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．そのときに，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場合，攻撃者によって SQL コマンドなどを注入され，データベース内のデータが盗み見や改ざんされうる．

たとえば，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．

2.3 クロスサイト・スクリプティング（XSS）

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．このときに，入力に対して適切にチェックを行っていない場合，XSS が生じうる．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，クッキーなどの情報を漏洩させる．

たとえば，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．このとき，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

© 2008 Information Processing Society of Japan

の Web ページをネットワークに出力している．クライアント側からスクリプト・コードを含む入力が行われたときに，

```
<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．

3. 動的な情報フロー追跡（DIFT）

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．たとえば，

```
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などへのアクセスや，ネットワークとの通信などが行われたときに，DIFT では，**表 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，条件を満たしたときに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，包括性があげられる．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．

表 1　DIFT の出力検査
Table 1　Checkings in DIFT.

出力先	テイント情報の検査対象	検出するアタック
ネットワーク	スクリプトタグ	Cross-site scripting
データベース	SQL コマンド	SQL injection
ファイル	公開ファイル以外のパス（システムコール open，execve など）	PHP remote file inc. Directory traversal Symbolic link follow. Command injection
シェル	メタ文字	Command injection

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，デコードなどの文字コード変換や，大文字小文字変換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・デコードなど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，テイント情報をすべ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，文献 5)，11)，12) などの既存の DIFT でも問題点としてあげられている．

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，ソースコード変換による手法で，オーバヘッドが大きい．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 の課題としてあげられる．

3.3 既 存 手 法

既存の DIFT 方式としては，インタプリタによる手法や，ソースコード変換による手法，ハードウェアによる手法が提案されている．以下で，それぞれの手法について述べる．

© 2008 Information Processing Society of Japan

```
$str =~ s/([^\w ])
        /'%'.unpack('H2', $1)/eg;
$str =~ tr/ /+/;
```

図 3　Perl による URL エンコード
Fig. 3　URL Encode in Perl.

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[1] 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．たとえば，PHP インタプリタ上に実現したものとして，Pietraszek らの手法[9]，Nguyen-Tuong らの手法[8] 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[6]，Livshits[7] らの手法などがある．インタプリタ方式では，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，各言語の命令単位で伝播させる．そして，インタプリタから出力されるときに検査を行う．

インタプリタ方式では，速度低下は 15%程度であり[7]，それほど大きくない．また，データの属性や，標準関数の操作などを把握しているため伝播の精度は比較的高い．しかし，間接的なデータの依存関係を根本的に解決しておらず十分とはいえない．たとえば，**図 3** は Perl による URL エンコードの典型的なコードである．このプログラムでは，変換時に暗黙的依存が存在するため，既存のインタプリタ方式では変換後のデータに伝播しない．また，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いる．そのため，新たな言語で記述されたプログラムに対しては，そのインタプリタに新たに実装する必要があり，包括性に欠ける．また，バイナリプログラムに対しては対応する手段がない．

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[11),12] が知られている．Xu らの手法では，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．Xu らの手法では，アドレス依存，暗黙的依存に対してアドホックな対策を行っている．この手法では，ポイント変数はつね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);
```

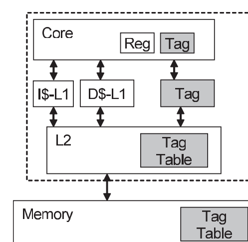


図 4　記憶領域に対するタグ
Fig. 4　Memory tag.

のようなコードでは伝播させることができず，十分とはいえない．また，暗黙的依存についても，Xu らの手法では，特定のパターンにのみ伝播させ不十分である．また，方式を実現するためにソースコードが必要であり包括性に欠け，さらに，速度低下も 80%程度[12] と大きい．

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，文献 4)，10) など，バッファ・オーバフローなど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[5] 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．

Raksha は，**図 4** のように，メモリ，レジスタ，キャッシュなど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を記録する．まず，入力データが取り込まれたメモリ領域のタグに，OS など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・オペランド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，出力時に，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，プログラムがどのインタプリタ上で動作しているかどうかや，ソースコードが入手可能であるかに依存しない．したがって，ほとんどすべ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

© 2008 Information Processing Society of Japan

表 2　DIFT
Table 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	○	○	△

分である．Raksha は，暗黙的依存に対してはまったく伝播を行わない．また，アドレス依存に関しては，前述したコードで，

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

のように，ソース・オペランドではなく，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．このような伝播はアドレス伝播と呼ばれているが，Raksha では，誤検出の多さから，標準ではアドレス伝播は行われていない．

3.3.4　ま と め

既存の DIFT をまとめると，**表 2** のようになる．インタプリタ方式は伝播精度は比較的高いが，包括的でない．ソースコード変換方式は，包括的でなく，実行速度のオーバヘッドも大きい．また，ハードウェア方式は，包括的かつ速度低下が小さいものの，伝播精度が十分でない．このように，既存の DIFT では，包括性，実行速度，伝播精度に関して十分とはいえない．

4. 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 と同様の方法をとる．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とで，包括的，かつ，実行速度のオーバヘッドを小さくする．また，Raksha とは異なり，命令のソース・オペランドからデスティネーション・オペランドに対して伝播を行わない．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ときに，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よらない伝播を行う．すなわち，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，直接的な依存と同様に対応可能にする．

表 3　メモリアクセス・パターン
Table 3　Memory access pattern.

コマンド	ld	ld	ld	st	st	ld	st	ld	st	ld	st	st	st	st
アドレス	40	32	33	14	64	20	65	34	52	35	76	34	66	67
サイズ	4	1	1	4	1	1	1	1	4	1	4	4	1	1
テイント	0	1	1	-	-	0	-	1	-	1	-	-	-	-

SWIFT では，まず，プログラムに対して外部から入力が行われた時に，その入力データが取り込まれたメモリ領域に対して，ソフトウェアでテイントと印付けする．そして，印付けされたテイント情報を伝播させるために，SWIFT では，まず，インオーダなメモリアクセス（ロードおよびストア）のパターンを，動的に取得する．そして，メモリアクセス・パターンに基づき，アクセスされたロードおよ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時に，移動元から移動先にテイント情報を伝播させる．そうして，出力時に，ソフトウェアでテイント情報の検査を行う．

4.2　文字列の識別

1 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見なす．たとえば，**表 3** のような，メモリアクセスが行われたとき，連続的に増加する 2 個以上の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，また，同様に，ストアアドレスの列 64〜67 を，同じ文字列要素へのストアと識別する．

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，文字列操作の特徴として，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あげられる．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時に，その 2 つの文字列間で移動が行われていると見なす．

たとえば，表 3 では，アドレス 32〜35 からなる文字列とアドレス 64〜67 からなる文字

© 2008 Information Processing Society of Japan

列は，データサイズ 2 バイトずつ，2 回以上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見なす．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．

5. SWIFT の詳細な動作

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．

5.1 ハードウェア構成

SWIFT では，メモリやキャッシュに対してバイトあ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 と同様の方法をとる．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．

また，テイント情報の伝播のために，**図 5** のように，Load String Table（LST），Store String Table（SST）という 2 つの CAM のテーブルを追加する．LST は，ロード命令時にその情報を格納するテーブルで，LST 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．また，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．SST 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．なお，図 5 の C1〜C9 は演算を行う回路であり，その演算の仕方については後述する．

5.2 文字列の識別

5.2.1 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[2] 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，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，そのストア命令のアドレス，データサイズと，SST の各エントリで記憶された文字列の情報を比較することで，ストアされた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．

5.2.2 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，各々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．

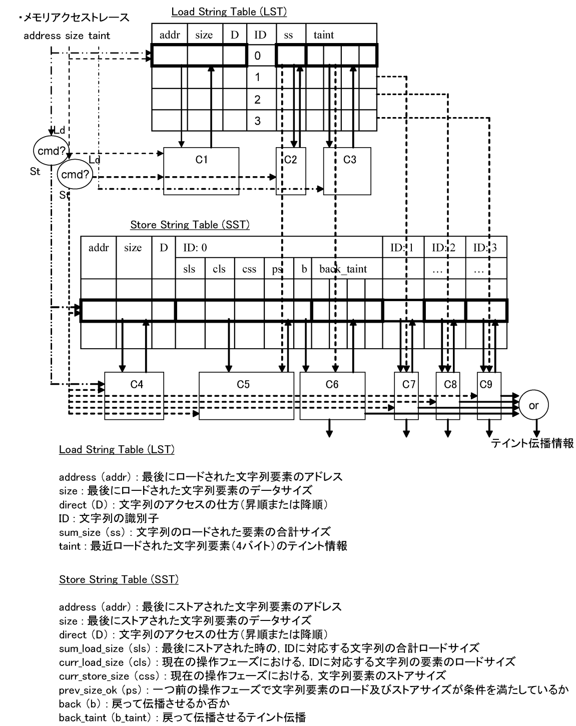


図 5　SWIFT のハードウェア構成
Fig. 5　Hardware structure of SWIFT.

ストア命令時に，そのストア命令の情報に従って，SST を更新する（図 5 (C4)）．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，その条件を満たすエントリに対応する文字

© 2008 Information Processing Society of Japan

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 6　文字列の識別
Fig. 6　String identification.

列の次の要素と見なす．そして，その条件を満たしたエントリの *address*，*direct*，*size* を更新する．*address* を *s_addr*，*size* を *s_size* とし，また，*direct* が 0 のとき，(1) では，*direct* を + に，(2) では，*direct* を − にする．なお，条件を満たすエントリが複数存在するときには，最近更新されたエントリを選択する．また，(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．

5.2.3　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（図 5 (C1)）．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，ロードされた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，テイント情報を持つ要素を含む文字列のみ識別される．

5.2.4　動　作　例

文字列識別の動作を例をあげて説明する．**図 6** (b)〜(e) は，図 6 (a) のようなメモリアクセスが行われたときの，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 6 (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 6 (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 6 (d)，(e) のように，先ほど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．

5.3　文字列操作の識別

5.3.1　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見なす．

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．たとえば，**図 7** (a) では，アドレス 32，33 のロード（ステップ 1）とアドレス 64，65 のストア（ステップ 2）を合わせて，1 つの操作フェーズであり，同様に，アドレス 34，35，66，67 へのアクセスも 1 つの操作フェーズとなる．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 SST 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，現在および 1 つ前の操作フェーズのロードおよ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，現在および 1 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．

5.3.2　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（図 5 (C5)，(C7)〜(C9)）．*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，1 つ前の操作フェーズのロードおよ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_load_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（図 5 (C2)），*sum_load_size* は，最後にストアされたときの，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 ブロックの *sum_load_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，SST のその ID ブロックの *sum_load_size* と，ID に対応する LST の *sum_size* の値が等しければ，同じ文

© 2008 Information Processing Society of Japan

(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	ID	ss	taint			
82	0	40	1	1	0	0
33	1	2			1	1

(b2) アドレス33ロード時のSST

addr	ID: 1								
	sls	cls	css	ps	b	b_taint			

(c1) アドレス65ストア時のLST

addr	ID	ss	taint			
82	0	40	1	1	0	0
33	1	2			1	1

(c2) アドレス65ストア時のSST

addr	ID: 1								
	sls	cls	css	ps	b	b_taint			
14	2	2	4	0	1	0	0	1	1
65	2	2	2	0	1	0	0	1	1

(d1) アドレス35ロード時のLST

addr	ID	ss	taint			
82	0	40	1	1	0	0
35	1	4	1	1	1	1

(d2) アドレス35ロード時のSST

addr	ID: 1								
	sls	cls	css	ps	b	b_taint			
14	2	2	4	0	1	0	0	1	1
65	2	2	2	0	1	0	0	1	1

(e1) アドレス66ストア時のLST

addr	ID	ss	taint			
82	0	40	1	1	0	0
35	1	4	1	1	1	1

(e2) アドレス66ストア時のSST

addr	ID: 1								
	sls	cls	css	ps	b	b_taint			
14	2	2	4	0	1	1	1	0	0
66	4	2	1	1	0	0	0	0	0

図 7　文字列操作の識別
Fig. 7　String operation identification.

字列の 1 つ前のストアと同じ操作フェーズであるとし，同じ ID ブロックの *curr_store_size* に，*s_size* を加算する．また，ある ID の *sum_load_size* と *sum_size* の値が異なれば，同じ文字列の 1 つ前のストアから操作フェーズが遷移したと見なす．その場合，1 つ前の変換フェーズのロードデータおよびストアデータのサイズを記録するために，同じ ID ブロックの *prev_size_ok* を，

- $prev_size_ok = (0 < curr_load_size \leq 4 \wedge curr_store_size \leq 4)?1:0$

とする．また，現在の操作フェーズのロードサイズおよび，ストアサイズなどを初期化する．すなわち，

- $curr_load_size = sum_size - sum_load_size$
- $curr_store_size = s_size$
- $sum_load_size = sum_size$

とする．

5.3.3　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，

- $prev_size_ok = 1 \wedge 0 < curr_load_size, curr_store_size \leq 4$

を満たすのなら，その条件を満たす 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見なし，現在および 1 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（図 5 (C6)〜(C9)）．

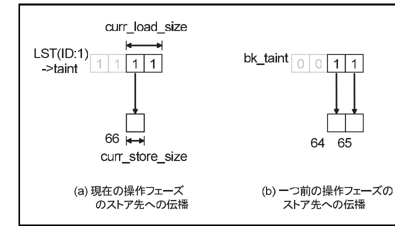


図 8　テイント情報の伝播
Fig. 8　Taint propagation.

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（図 5 (C3)），LST の *taint* を参照し，**図 8** (a) のように，テイント情報をストア先に伝播する．また，1 つ前の操作フェーズのストア先への伝播には，1 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 *back_taint* を用いる．*back_taint* を参照し，図 8 (b) のように，テイント情報を伝播させる．

5.3.4　動　作　例

文字列操作識別の様子を，例をあげて説明する．図 7 (b)〜(e) は，図 7 (a) のようなメモリアクセスが行われたときの，LST の各項目と，SST における ID: 1 のブロックの各項目の更新の様子である．アドレス 33 のロード時に，アドレス 32 を基点とした文字列（以下，文字列 I）のロードされた合計サイズは 2 バイトであるので，図 7 (b) のように，*sum_size* は 2 となる．また，*taint* にロード・データのテイント情報が記録される．また，アドレ

© 2008 Information Processing Society of Japan

ス 65 のストア時，文字列 I では，すでに 2 バイトロードされているので，図 7 (c) のように，*sum_load_size*，*curr_load_size* は 2 になる．また，同時に，*curr_store_size*，*back*，*back_taint* も更新される．さらに，アドレス 35 のロード時に，図 7 (d) のように，*sum_size*，*taint* が更新される．そして，アドレス 66 のストア時には，文字列 I では，新たに，2 バイト，ロードされたので，操作フェーズが遷移し，図 7 (e) のようにエントリが更新される．また，そのエントリの各項目から，アドレス 34 からアドレス 66 にテイント情報が伝播される．また，*back_taint* に基づいて，アドレス 32，33 からアドレス 64，65 にも伝播される．

5.4 議　　論

5.4.1 誤検出，検出漏れ

SWIFT では連続的に 2 回以上行われるメモリ・アクセスを文字列と見なす．したがって，連続的なアクセスが行われない文字列は検出できず，その結果，検出漏れが生じる．また，文字列以外で連続領域へのアクセスされると文字列と見なされる．したがって，その連続領域にテイント情報が伝播し，さらに，そのテイント情報が他の文字列へ伝播した場合に，誤検出となる．また，文字列のサイズが 4 バイト以下のとき，2 回以上のアクセスが行われず，検出漏れを引き起こす．ただし，これに関しては，一般的にアタックには一定以上のサイズが必要とされ，それほど問題にならない．また，SWIFT では，文字列が交互に 1〜4 バイトずつアクセスされることを文字列操作と見なし，テイント情報を伝播させる．したがって，4 バイトを超えたアクセスが文字列操作で行われると検出漏れを引き起こす．また，移動元と移動先のデータサイズが異なる場合に，文字列操作の最後の操作フェーズで移動元と移動先を正確に判定できず，伝播の精度が下がる．さらに，LST，SST のエントリ数が不足すると，検出漏れが起こる．

5.4.2 速度のオーバヘッド

SWIFT では，ロードおよびストア命令実行時に処理を行う．ロード時に，ロード領域のタグを参照し，そして，LST の 1 つのエントリを更新する．また，ストア時に，SST の 1 つのエントリと，LST のすべてのエントリを参照し，SST の更新とストア領域のタグへの代入を行う．タグはハードウェアとして追加されるので，既存のハードウェア方式[10) と同様に，タグの参照，代入によるオーバヘッドはそれほど生じない．また，LST および SST のエントリ数は，各々4，64 と想定しているので，LST，SST の更新によるオーバヘッドも小さい．ただし，今後，エントリ数を大きくする必要が生じた場合には，そのオーバヘッドを考慮しなければならない．

表 4　LST の項目のビット数
Table 4　LST.

LST の項目	ビット数
address（addr）	32
size	3
direct（D）	2
ID	3
sum_size（ss）	8
taint	4
合計	52

表 5　SST の項目のビット数
Table 5　LST.

SST の項目	ビット数
address（addr）	32
size	3
direct（D）	2
sum_load_size（sls）	8×4
curr_load_size（cls）	3×4
curr_store_size（css）	3×4
prev_size_ok（ps）	1×4
back（b）	1×4
back_taint（b_taint）	4×4
合計	149

5.4.3 ハードウェア量のオーバヘッド

ハードウェア量のオーバヘッドの大部分はタグである．既存のハードウェア方式[10) と同様に，SWIFT では，キャッシュや主記憶のバイトデータあたりのタグを管理する機構が必要である．また，SWIFT では，そのほかに，テイント情報伝播のため，LST，SST というテーブルが必要である．Raksha などのハードウェア方式では，基本的に，伝播には，OR 演算を行う機構があればよいので，SWIFT は，既存手法と比べて，ハードウェア容量は大きくなる．LST の各エントリのビット数は，**表 4**，**表 5** のように 52 bit，SST の各エントリのビット数は，LST のエントリ数を 4 とした場合，149 bit になる．

© 2008 Information Processing Society of Japan

6. 評　　価

6.1 方　　法

SWIFT の評価は，x86 エミュレータ Bochs 3.5 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 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，特に記載がない場合，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 結　　果

6.2.1 文字列操作

まず，**表 6** のような PHP の文字列操作関数を用いて，伝播精度の評価を行った．(a)〜(d) は，抽出，結合，挿入，照合という典型的な文字列操作であり，(e)〜(k) 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・デコードという，ネットワーク・アプリケーションで特によく使われる文字列変換である．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 8** 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．

Raksha_n，Raksha_a，SWIFT を実行し，評価を行った．その結果，**表 7**，表 8 のようになった．表 8 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，テイント情報を持つ箇所である．(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，すべ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，すべ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・デコードではすべ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，すべ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，すべ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．

また，SWIFT の LST，SST のエントリ数を変えて評価を行った．LST のエントリ数を 2，SST のエントリ数を 16，32 とした．その結果，**表 9** のようになった．(a)〜(c)，(e)〜(j) では，すべてのエントリ数で伝播可能であったが，(k) では，LST のエントリ数を 2，SST

表 6　文字列操作関数による評価
Table 6　Evaluations by string operation functions.

	関数	言語	操作
(a)	substr	PHP	抽出
(b)	”.” 演算子	PHP	結合
(c)	ereg_replace	PHP	挿入
(d)	ereg	PHP	照合
(e)	strtoupper	PHP	大文字変換
(f)	strtolower	PHP	小文字変換
(g)	urlencode	PHP	URL encode
(h)	urldecode	PHP	URL decode
(i)	base64_encode	PHP	base64 enc.
(j)	base64_decode	PHP	base64 dec.
(k)	図 3	Perl	URL encode

表 7　文字列操作関数による評価結果
Table 7　Results of evaluations by string operation functions.

	伝播の有無		
	Raksha_n	Raksha_a	SWIFT
(a)	○	○	○
(b)	○	○	○
(c)	○	○	○
(d)	−	−	−
(e)	×	○	○
(f)	×	○	○
(g)	×	○	○
(h)	○	○	○
(i)	×	○	○
(j)	×	○	○
(k)	×	×	○

© 2008 Information Processing Society of Japan

表 8　文字列操作関数による評価結果（詳細）
Table 8　Results of evaluations by string operation functions in detail.

	入力	正しい依存の仕方	Raksha_n による伝播	Raksha_a による伝播	SWIFT による伝播
(a)	abcdefrhirklmno	defrhirk	–	–	–
(b)	abcde	PPPabcdePPP	–	–	–
(c)	abcdefrhirklmno	abcdePPPPPklmno	–	–	–
(d)	tonight	tomorrow	–	–	–
(e)	abcdefrhi	ABCDEFRHI	ABCDEFRHI	–	–
(f)	abcdefrhi	abcdefrhi	abcdefrhi	–	–
(g)	a<b>c;	a%3Cb%3Ec%3B	a%3Cb%3Ec%3B	a%3Cb%3Ec%3B	a%3Cb%3Ec%3B
(h)	a%62+%64e%66r%68i	ab defrhi	ab defrhi	ab defrhi	–
(i)	abcdefghi	YWJjZGVmZ2hp	YWJjZGVmZ2hp	–	YWJjZGVmZ2hp
(j)	YWJjZGVmZ2hpamts	abcdefghijkl	abcdefghijkl	–	–
(k)	a<b>c;d:	a%3Cb%3Ec%3Bd%3a	a%3Cb%3Ec%3Bd%3a	a%3Cb%3Ec%3Bd%3a	a%3Cb%3Ec%3Bd%3a

表 9　LST，SST のエントリ数変更による評価
Table 9　Changing entry of LST and SST.

	LST:2	SST:16	SST:32
(a)	○	○	○
(b)	○	○	○
(c)	○	○	○
(e)	○	○	○
(f)	○	○	○
(g)	○	○	○
(h)	○	○	○
(i)	○	○	○
(j)	○	○	○
(k)	×	×	○

のエントリ数を 16 とした場合に伝播させることができなかった．

6.2.2　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 10** 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，すべての手法で，表 1 のように行った．

その結果，表 10 のようになった．すべての手法ですべてのアタックを検出することができた．また，SWIFT および，Raksha_n では誤検出が生じなかった．**表 11** 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．すべてのプログラムで，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，すべてのプログラムで誤検出が生じた．

6.3　考　　察

6.3.1　既 存 手 法

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，現実的な手法とはいえない．このアドレス依存による誤検出は，文献 5)，11) 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すべ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，入力データをすべ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

© 2008 Information Processing Society of Japan

表 10　ネットワーク・アプリケーションによる評価結果
Table 10　Results of evaluations by network applications.

CVE	Program	Attack type	Raksha_n		Raksha_a		SWIFT	
			検出	誤検出	検出	誤検出	検出	誤検出
2005-0870	phpsysinfo 2.3	Cross site scripting	あり	なし	あり	あり	あり	なし
2003-0486	phpBB 2.0.5	SQL injection	あり	なし	あり	あり	あり	なし
2006-0983	Qwikiwiki 1.4.1	Directory traversal	あり	なし	あり	あり	あり	なし
2005-2380	phpSurveyor 0.98	Cross site scripting	あり	なし	あり	あり	あり	なし
2005-2398	phpSurveyor 0.98	SQL injection	あり	なし	あり	あり	あり	なし
2003-1435	PHPnuke 6.0	SQL injection	あり	なし	あり	あり	あり	なし

表 11　ネットワーク・アプリケーションの評価結果（詳細）
Table 11　Results of evaluations by network applications in detail.

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>
2005-2398	sid=1;DELETE * FROM table
2003-1435	<= 1000;DELETE * FROM table ORDER

は，本評価ですべてのアタックが検出されたとはいえ，安全とはいいがたい．

また，ソフトウェア・ベースの DIFT でも，文字列操作関数で間接的依存が存在しているために伝播できない場合がある．インタプリタ方式では，表 6 (k) で間接的依存が生じているため，検出漏れを引き起こす．また，ソースコード変換方式では，表 6 (e)〜(g)，(i)〜(k) で，間接的依存が生じているため，伝播の精度を下げる要因となっている．

6.3.2 SWIFT

一方，SWIFT では，本評価で試したアタックを誤検出なくすべ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いえる．しかし，SWIFT にも問題点がある．表 6 (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 (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は，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．また，LST と SST のエントリ数については，それぞれ 4，64 とすることで，今回，評価したプログラムでは十分であった．

7. おわりに

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従って，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．

今後の課題としては，伝播精度の向上があげられる．文字列アクセスや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，速度およびハードウェア容量のオーバヘッドについて，ハードウェア・シミュレータなどを用いた定量的な評価も求められる．さらに，命令単位で伝播を行う既存のハードウェア・ベースの DIFT との協調に関しても検討したい．

謝辞　本稿の研究は，一部，半導体理工学研究センター（STARC）および，JST CREST，科学研究費補助金（特定領域研究）No.19024020 による．

参　考　文　献

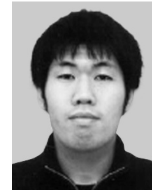
1) Allen, J.: Perl version 5.8.8 documentation — perlsec (2006). http://perldoc.perl.org/perlsec.pdf
2) Baer, J.L. and Chen, T.F.: Effective hardware-based data prefetching for high-performance processors, *IEEE Computer Society*, Vol.44, No.5, pp.609–623 (1995).
3) Christey, S. and Martin, R.A.: Vulnerability type distributions in cve (2007).

© 2008 Information Processing Society of Japan

http://cve.mitre.org/docs/vuln-trends/index.html
4) Crandall, J. and Chong, F.: Minos: Control data attack prevention orthogonal to memory model, *37th International Symposium on Microarchitecture* (*MICRO*), pp.221–232 (2004).
5) Dalton, M., Kannan, H. and Kozyrakis, C.: Raksha: A flexible information flow architecture for software security, *34th International Symposium on Computer Architecture* (*ISCA*), pp.482–493 (2007).
6) Haldar, V., Chandra, D. and Franz, M.: Dynamic taint propagation for java, *21st Annual Computer Security Applications Conference* (*ACSAC*), pp.303–311 (2005).
7) Livshits, B., Martin, M. and Lam, M.S.: Securifly: Runtime protection and recovery from web application vulnerabilities, Technical report, Stanford Univrsity (2006).
8) Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J. and Evans, D.: Automatically hardening web applications using precise tainting, *20th IFIP International Information Security Conference* (*SEC*), pp.295–307 (2005).
9) Pietraszek, T. and Berghe, C.V.: Defending against injection attacks through context-sensitive string evaluation, *8th International Symposium on Recent Advances in Intrusion Detection* (*RAID*), pp.124–145 (2005).
10) Suh, G., Lee, J. and Devadas, S.: Secure program execution via dynamic information flow tracking, *11th International Conference on Architectural Support for Programming Languages and Operating System* (*ASPLOS-XI*), pp.85–96 (2004).
11) Xu, W., Bhatkar, S. and Sekar, R.: Practical dynamic taint analysis for countering input validation attacks on web applications, Technical Report SECLAB-05-04 (2005).
12) Xu, W., Bhatkar, S. and Sekar, R.: Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks, *15th USENIX Security Conference*, Vol.15, pp.121–136 (2006).

(平成 20 年 1 月 29 日受付)
(平成 20 年 5 月 9 日採録)

勝沼　　聡（正会員）
1983 年生．2006 年東京大学工学部電子情報工学科卒業．2008 年同大学大学院情報理工学系研究科修士課程修了．同年より株式会社日立製作所中央研究所勤務．ディペンダブルコンピューティング等の研究に従事．



塩谷　亮太（学生会員）
1981 年生．2008 年東京大学大学院情報理工学系研究科修士課程修了．同研究科博士課程に在学中．コンピュータアーキテクチャの研究に従事．

入江　英嗣（正会員）
1999 年東京大学工学部電子情報工学科卒業．2004 年同大学大学院情報理工学系研究科博士課程修了．博士（情報理工学）．同年より 2008 年まで科学技術振興機構 CREST “ディペンダブル情報基盤” 研究員．2008 年より東京大学情報理工学系研究科助教．プロセッサアーキテクチャの研究に従事．電子情報通信学会，ACM 各会員．

五島　正裕（正会員）
1968 年生．1992 年京都大学工学部情報工学科卒業．1994 年同大学大学院工学研究科情報工学専攻修士課程修了．同年より日本学術振興会特別研究員．1996 年京都大学大学院工学研究科情報工学専攻博士後期課程退学，同年より同大学工学部助手．1998 年同大学大学院情報学研究科助手．博士（情報学）．2005 年東京大学情報理工学系研究科助教授，2007 年同大学同研究科准教授，現在に至る．コンピュータ・システムの研究に従事．著書に『ディジタル回路』．2001 年情報処理学会山下記念研究賞，2002 年同学会論文賞受賞．IEEE 会員．

© 2008 Information Processing Society of Japan

坂井　修一（正会員）

1981 年東京大学理学部情報科学科卒業．1986 年同大学大学院工学系研究科修了，工学博士．電子技術総合研究所，MIT，RWC，筑波大学を経て，1998 年東京大学助教授．2001 年より東京大学大学院情報理工学系研究科教授．現在同研究科副研究科長．計算機システムおよびその応用の研究に従事．特に並列処理アーキテクチャ，相互結合網，最適化コンパイラ，省電力アーキテクチャ，ディペンダブルシステム等の研究を進めている．情報処理学会研究賞（1989），情報処理学会論文賞（1991），IBM 科学賞（1991），市村学術賞（1995），IEEE Outstanding Paper Award（1995），Sun Distinguished Speaker Award（1997）等受賞．電子情報通信学会，人工知能学会，ACM，IEEE 各会員．日本学術会議連携会員，日本学術振興会専門委員．

© 2008 Information Processing Society of Japan