情報処理学会論文誌 Vol.53 No.9 2075–2085 (Sep. 2012)

コールスタックの制御データ検査による スタック偽装攻撃検知

冨永 悠生[1,a)] 樫山 武浩[1] 瀧本 栄二[1] 桑原 寛明[1] 毛利 公一[1]
齋藤 彰一[2] 上原 哲太郎[3] 國枝 義敏[1]

受付日 2011年11月30日, 採録日 2012年6月1日

概要：既存のホスト型侵入検知システムやコンパイラの拡張によるバッファオーバフローの検知では，検知システムを回避した攻撃が存在し問題となっている．我々は検知システムを回避する攻撃の１つであるスタック偽装攻撃に着目し，スタック偽装攻撃を検出することを目的としたこれまでにない侵入検知システムを新たに提案する．本システムでは，コールスタックに積まれたフレームポインタとリターンアドレスからなる制御データを検査し，制御データの書き換えを検知することでスタック偽装攻撃を防ぐ．関数呼び出し時に制御データをバッファオーバフローによる書き換えを受けないメモリ領域に退避させ，関数呼び出し元への復帰時に退避させた制御データとスタック上の制御データが一致するかを検査する．これにより，スタック偽装攻撃による不正な制御データの偽装を検知できる．提案手法を実装し，gzip と httpd ならびに wc を動作させた際，本提案による増加部分が全実行時間に占める割合は，それぞれ 2.52%，71.09%，94.82%であった．

キーワード：スタック偽装攻撃，侵入検知システム，バッファオーバフロー，脆弱性

Mimicry Attack Detection by Saving and Checking Control Data Stored on Call Stack

Yuuki Tominaga[1,a)] Takehiro Kashiyama[1] Eiji Takimoto[1]
Hiroaki Kuwabara[1] Koichi Mouri[1] Shoichi Saito[2] Tetsutaro Uehara[3]
Yoshitoshi Kunieda[1]

Received: November 30, 2011, Accepted: June 1, 2012

Abstract: There are many methods to detect buffer overflow using host-based intrusion detection systems or compiler extensions. Some attacks, however, can avoid these defense systems. One of such attacks is “mimicry attack”. In this paper, we propose a new intrusion detection system focusing on mimicry attack. Our system detects invalid control data (frame pointer and return address on call stack) overwritten by mimicry attack. This detection method consists of saving and checking processes. The saving process, which is invoked when entering every functions, saves control data to the invulnerable memory area. The checking process, which is invoked when exiting any functions, compares the saved and real control data. We have implemented this proposed system and evaluated its time overhead. The percentage of the processing time for this proposed method in each total execution time of gzip process, httpd and wc are 2.52%, 71.09% and 94.82% respectively.

Keywords: mimicry attack, intrusion detection system, buffer overflow, software vulnerabilities

[1] 立命館大学
Ritsumeikan University, Kusatsu, Shiga 525–8577, Japan
[2] 名古屋工業大学
Nagoya Institute of Technology, Nagoya, Aichi 466–8555, Japan
[3] NPO 情報セキュリティ研究所
The Research Institute of Information Security, Tanabe, Wakayama 646–0011, Japan
[a)] y-tominaga@hpcss.is.ritsumei.ac.jp

© 2012 Information Processing Society of Japan

1. はじめに

バッファオーバフロー脆弱性を代表とするプログラムの脆弱性を突いた攻撃は，プログラムの制御フローを開発者の意図に反するものに変える．特に，高い権限で動作しているプログラムに対する攻撃は高い権限のまま実行されるため，システムの乗っ取りやデータの改ざんなどの甚大な被害が発生する．脆弱性が存在しうるプログラムを安全に動作させる手段としてホスト型侵入検知システム [1] がある．ホスト型侵入検知システムは，プログラムの動作があらかじめ定義された規則に従っているかをプログラムの実行時に検査する．攻撃コードに由来する規則に反する動作を検知でき，プログラムを安全に動作させることができる．

しかし，Wagner らは，正当なシステムコールを順に発行しながら悪意ある動作を実行する攻撃コードを作り上げることでホスト型侵入検知システムの検知が回避できることを示している [2]．彼らはこのような攻撃を Mimicry Attack（以下，本論文ではスタック偽装攻撃と呼ぶ）と命名している．さらに，Kruegel らは，静的なバイナリ解析に基づいてスタック偽装攻撃を自動化する方法について提案しており，スタック偽装攻撃が容易に起こりうることを示している [3]．

本論文では上記のスタック偽装攻撃を防ぐことを目的として，関数呼び出しごとにコールスタックに積まれるフレームポインタとリターンアドレスの値（以後，本論文ではこれらの値を「制御データ」と呼ぶ）を検査してスタック偽装攻撃を検知する手法を提案する．

以下，2 章で既存のバッファオーバフロー脆弱性を用いた攻撃の対策手法と本論文で問題とするスタック偽装攻撃について記述しなおし，3 章で提案手法，4 章で実装方法についてそれぞれ具体的に述べ，5 章で評価，6 章で考察し，7 章でまとめる．

2. 既存の対策手法とスタック偽装攻撃

本章では，まず 2.1 節で既知のプログラムの脆弱性について概説する．次に，2.2 節では，その脆弱性を突く攻撃の代表例をあげ解説し，2.3 節では，それらの攻撃の対策について要約する．最後に，2.4 節では，そうした対策をすり抜けるスタック偽装攻撃について詳述する．なお，ここで取り上げているスタックフレームに関する記述は，現行の x86 アーキテクチャ向けに linux 上の GCC などで採用されている最も代表的な形式を前提としている．

2.1 プログラムの脆弱性

既知のプログラムの脆弱性 [4] として，スタック領域とヒープ領域を標的としたバッファオーバフロー脆弱性と printf(3) 関数で発生しうる書式文字列（Format string attack）[5], [6] の脆弱性があげられる．まず前者のバッファオーバフロー脆弱性は，用意されたメモリ領域よりもデータサイズが大きい場合に，用意されたメモリ領域を越えてデータが書き込まれることに由来する．

この脆弱性を突き，どのように攻撃が行われるかについては次節で詳述する．一方，書式文字列における脆弱性は，文字列の表示方法を決定する書式文字列を攻撃者が操作できる場合に発生する．詳細は文献 [6] 参照．

2.2 プログラムの脆弱性を突いた攻撃手法

前節で述べたプログラムの脆弱性を利用し，攻撃者はヒープ領域を書き換えることもありうるが，以下では，より影響が大きいスタックフレームの書き換えに焦点を当てる．すなわち，コールスタック上に存在するフレームポインタやリターンアドレスを任意の値に書き換えることで，結果的に攻撃者はシステムの乗っ取りやデータの改ざんすらも可能となる．

こうした攻撃の典型例がコードインジェクション攻撃と Return-into-libc 攻撃である．それぞれの攻撃方法について図 1 を用いて説明する．

コードインジェクション攻撃

コードインジェクション攻撃は図 1 (1) に示すようにプログラムの脆弱性を利用して攻撃コードの挿入とリターンアドレスの書き換えを行う．リターンアドレスを攻撃コードの先頭アドレスを指し示すように書き換えることで，現在実行中の関数からの復帰時に，攻撃コードが実行されてしまう．

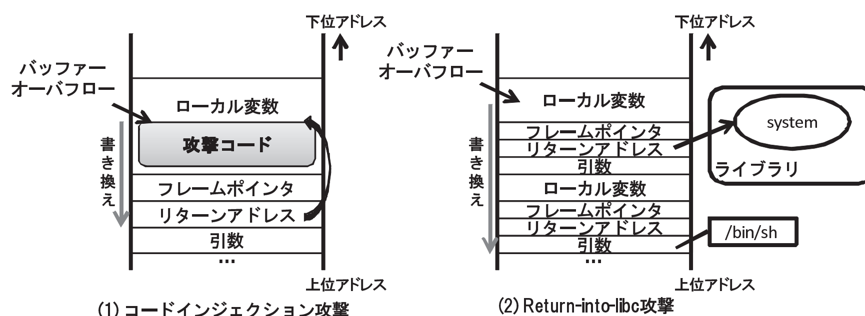

図 1　コードインジェクション攻撃と Return-into-libc 攻撃時のスタックフレーム
Fig. 1　Stack frame layout of Code Injection and Return-into-libc attacks.

© 2012 Information Processing Society of Japan

Return-into-libc 攻撃

Return-into-libc 攻撃は図 1 (2) に示すように実行中の関数のリターンアドレスを攻撃者が実行させたい任意の関数の先頭アドレスに書き換えるとともに，その関数の処理に必要なスタックフレームを用意することで，攻撃者の所期の関数を自在に動作させる攻撃である．たとえば，実行中の関数のリターンアドレスをライブラリ libc に含まれる system 関数のアドレスに書き換え，スタックフレームの引数に実行したいプログラムを指すパス文字列（たとえば`/bin/sh`）のポインタを上書きすることで，シェルの起動など目的の動作を実行させることができる．この攻撃では，既存のライブラリ関数などを使い攻撃を行うため，攻撃コードの挿入を行う必要はない点でコードインジェクション攻撃と異なる．

2.3 既存の攻撃検知機構

前節で述べた以外の未知のものも含め，スタックフレーム，特にリターンアドレスの書き換えによる攻撃を検知するための機構として，Executable space protection [7], [8]，StackGuard [9] や ProPolice [10] などの攻撃耐性システムならびにシステムコールに着目するなど様々な視点から検査するホスト型侵入検知システム [11], [12], [13], [14], [15], [16], [17], [18], [19], [20] が提案されているが，本節では代表的なものだけをあげる．

Executable space protection

Executable space protection は，CPU や OS のメモリ保護機能などとの連携により，コールスタック領域でのプログラムの実行を不可能にする．この手法では，コードインジェクションによってコールスタックに挿入されたコードの実行を防ぐことが可能である．

攻撃耐性システム

StackGuard は，GCC（GNU Compiler Collection）の拡張機能として実装されているリターンアドレスの書き換えを検知するための機構である．図 **2** (1) に StackGuard のスタックフレームを表す．StackGuard は，関数の呼び出し時にコールスタックのリターンアドレス直前にカナリアと呼ばれる値を挿入するコードと，関数呼び出し元への復帰処理時にカナリアの値が変更されていないか検査するコードをコンパイル時に生成する．プログラムの実行時にカナリアの値が変更されている場合，攻撃者によるリターンアドレスの書き換えが発生したと判断し，プログラムを強制的に終了させる．同様の機構は Microsoft Visual Studio においてもコンパイルオプション/GS として実装されている [21]．

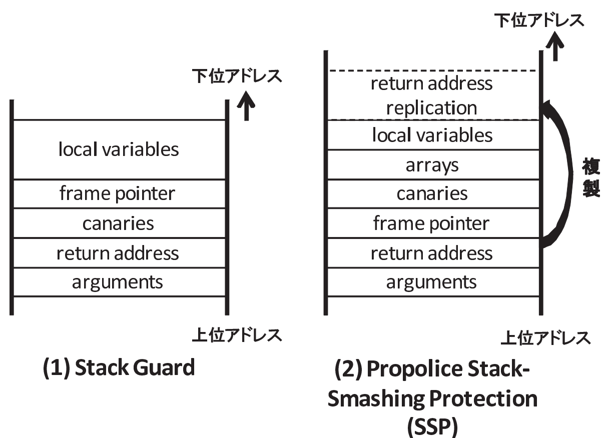

図 2 StackGuard と SSP のスタックフレームレイアウト
Fig. 2 Stack frame layout of StackGuard and SSP.

ProPolice は，StackGuard を改良して検知精度の向上とオーバヘッドの削減を実現する．ProPolice では，カナリアの挿入位置をリターンアドレス直前からフレームポインタの直前へと変更することで，リターンアドレスに加えてフレームポインタも保護する．バッファオーバフローの可能性がある配列を関数ポインタを含む局所変数領域よりも下位アドレスに配置することで，バッファオーバフローによる関数ポインタや局所変数の書き換えを防ぐ．さらに，ProPolice ではリターンアドレスをコールスタックの最上位にコピーし，リターンアドレスが書き換えられているかをコピーした値を用いて検査することでカナリアを回避する攻撃にも対処している．

これらの攻撃耐性システムでは，カナリアが書き換えられるコードインジェクション攻撃と Return-into-libc 攻撃の両方を防ぐことが可能である．しかし，書式文字列脆弱性を突く攻撃によりカナリアを回避して書き換えが行われる場合は攻撃を防ぐことができない．

ホスト型侵入検知システム

ホスト型侵入検知システムの多くは，プロセスが発行するシステムコールを監視し，不正なシステムコール発行を検知することで悪意あるコードの実行を防ぐ．静的解析情報のみを用いるシステム [11], [12], [13] は，ソースコードやバイナリからプログラムの挙動を静的に解析し，その挙動から逸脱することなく動作していることを実行時に検査する．Wagner らのシステム [11] では，プログラムのソースコードを解析して正常な動作時のシステムコール発行順序を状態遷移グラフとして定義し，実行時にシステムコールが発行されるたびにシステムコールの発行順をこの状態遷移グラフに基づいて検査する．一般にシステムコールの発行順序は非決定的なグラフとして表現されるため，実行時検査のオーバヘッドが大きい．静的解析情報と実行時情報を併用するシステム [15], [16], [17], [18], [19], [20] は，静的解析によって得られた挙動と実際にプログラムを動作させた際に得られるコールスタックやヒープ領域などの実行時情報を用いて実行時の挙動を検査し，不正な挙動が行われていないかを検査する．槇本らのシステム [19] は，ライブラリ関数の呼び出しごとにライブラリ関数の呼び出し順が静的解析によって得られた情報から逸脱していないかを

© 2012 Information Processing Society of Japan

検査する．また，システムコールが発行されるごとに静的解析によって得られた情報とシステムコール発行時点でのコールスタックの実行時情報を用いて関数呼び出しの順序が正常であるかを検査する．

2.4 スタック偽装攻撃

スタック偽装攻撃とは，2.2 節で述べた以外の未知の攻撃も含め，2.2 節で述べたスタックフレームの書き換えを大規模に行う攻撃である．実行中の関数のスタックフレーム以下のコールスタックに積まれている全スタックフレームを，プログラムが正常に動作しているときにありうるコールスタックとなるように書き換えることで，攻撃後のスタックフレームの状態を，正常なスナップショットに見せかける．スタック偽装が行われるとコールスタック上の関数呼び出し順は正常に見えるため，多くのホスト型侵入検知システムは正しく関数呼び出しが行われてきたと判断し，攻撃の見逃しが発生する．

スタック偽装攻撃を図 **3** を用いて説明する．図 3 の関数呼び出しは，プログラムの正常な動作として Func1→Func2→Func3→Func4 または Func1→Func5→Func6 の順で関数が呼び出されることを表している．関数が Func1→Func2→Func3→Func4 と呼び出されると，コールスタックには (1) のようにスタックフレームが積み上げられる．このとき，Func4 内にバッファオーバフロー脆弱性があり，それを突いてバッファオーバフローが発生すると，Func4 の局所変数領域より上位アドレスのローカル変数以下のコールスタックは任意に書き換えられる．もし，(2) のように関数が Func1→Func5→Func6 と呼び出されたときと同様のコールスタックに書き換えられた場合，Func4 の関数呼び出し元への復帰先は，偽装されたコールスタックから Func5 となる．Func4 から Func5 へと処理が移った後の動作はプログラム上の正常な動作としてありうる動作であり，通常得られる実行時情報を調べたとしても，Func4 から Func5 へと処理が移った証拠が残らないため攻撃の発生をこの後の動作から検知することは難しい．

既存の侵入検知システムは，システムコールごとに関数呼び出し順が正常であるかをコールスタックを用いて検査するため，システムコールを発行しない関数などでスタック偽装攻撃が発生した場合，攻撃を防ぐことは検査のタイミングとしても通常不可能である．

本論文は，上述のようにホスト型侵入検知システムを回避し任意の攻撃を行うスタック偽装攻撃からプログラムを守ることを目的とする．

3. コールスタックの制御データ検査によるスタック偽装攻撃検知

本章では，スタック偽装攻撃を防ぐための手法を提案する．スタック偽装攻撃は，前節で既述のとおりコールスタックを目的の関数が適切な順序で呼び出された時点のコールスタックに書き換えることで，正常な動作に偽装しつつ目的の関数を呼び出す．コールスタックの書き換えが発生すると，コールスタックに積み上げられた制御データが変更される．そこで，関数呼び出し元へと復帰する直前にコールスタックに積み上げられたすべての制御データを検査することで，制御データの書き換えを検知し，書き換えられた制御データに基づいてプログラムが動作することを防止できる．

ここでは，ユーザ関数内で発生するスタック偽装攻撃を防ぐことを目的とし，以下の 2 点からライブラリ内での検査は行わない．(1) ライブラリ内に存在する脆弱性はセキュリティ専門家の指摘によりすぐに修正されるため，ユーザプログラムより脆弱性が存在する可能性は少ない．(2) 検査対象範囲とオーバヘッドはトレードオフの関係であり，関数呼び出しの多いライブラリを検査対象範囲に含むことでオーバヘッドが極端に大きくなる．

提案手法の概要を図 **4** に示す．提案手法は「制御データの退避処理」と「制御データの検査処理」の 2 つの処理から構成される．関数呼び出し時に制御データの退避処理，関数呼び出し元への復帰時に制御データの検査処理を行うことでスタック偽装攻撃を検知する．

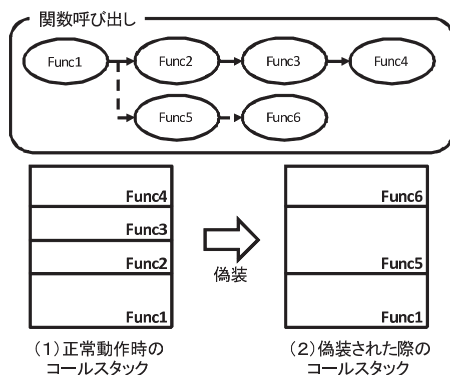

図 3 スタック偽装攻撃例
Fig. 3 An example of mimicry attack.

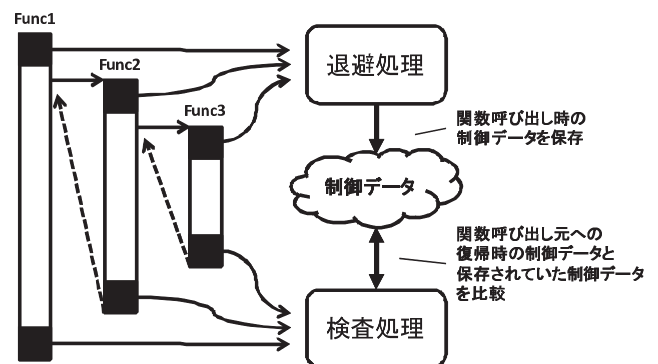

図 4 提案手法の概要
Fig. 4 Overview of our proposed method.

© 2012 Information Processing Society of Japan

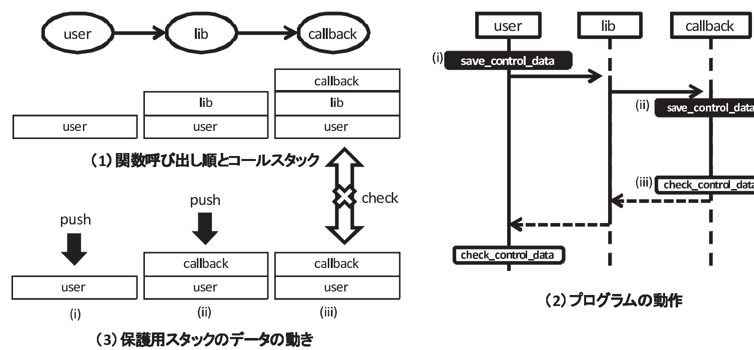

図 5 コールバック関数の問題

Fig. 5 A callback function problem.

制御データの退避処理

制御データの退避処理は，コールスタックに積まれたすべての制御データを別途用意したメモリ領域に退避させる．退避先には，カーネル空間とユーザ空間が考えられるが，たとえば 2.3 節であげた ProPolice の SSP のようにユーザ空間に制御データを退避させる場合は完璧なスタック偽装により，それすらも書き換えが発生しうることを考慮しなくてはならない．したがって，提案手法では，書き換えられる心配のないカーネル空間に制御データを退避させる．なお，以下では退避先の領域を保護用スタックと呼ぶ．また，これまで単にスタックと記してきた通常のスタックを，以降ではコールスタックと呼ぶ．

制御データの退避処理は関数呼び出しの直後に行う．この時点では，スタック偽装攻撃によるコールスタックの変更が行われることはないため，つねに関数呼び出し時の制御データを安全に退避させることが可能である．

保護用スタックに制御データを退避させる方法として，「呼び出された関数の制御データのみを保護用スタックに積み上げる方法」もしくは「コールスタック上のすべての制御データを保護用スタックに積み上げる方法」の 2 つが考えられる．ライブラリ関数呼び出しにおいては退避処理を行わないため，前者の方法ではライブラリ関数の制御データは保護用スタックに積まれない．そのため，コールバック関数のようにライブラリ関数からユーザ関数が呼び出される場合，**図 5** に示すようにコールスタックに積まれた制御データと保護用スタックに積まれた制御データが異なる問題が発生する．

図 5 の (1) はユーザ関数 user → ライブラリ関数 lib → ユーザ関数 callback の順に関数呼び出しが行われることとそれぞれの関数が呼び出された時点のコールスタック，(2) は (1) の関数呼び出しにおける提案手法の処理順，(3) は (2) の提案手法の処理順に対応した保護スタックの状態を示している．(2) の (iii) における制御データの検査処理は，(1) の callback におけるコールスタックと (3) の (iii) に示す保護用スタックを比較し，等しいか検査する．コールバック関数が呼び出される場合，退避された制御データがコールスタック上の制御データと一致せず異常であると判断し誤検知を引き起こす．そこで提案手法では，コールバック関数で発生する誤検知を解消するために，コールスタック上のすべての制御データを毎回保護用スタックに積み上げる方法を採用する．しかし，関数呼び出しごとに毎回コールスタックに含まれるすべての制御データをそのまま単純に保護用スタックに積み上げると，保護用スタックに積み上げる制御データの総数は関数呼び出しの深度より速く増加する．そのため，特に再帰によって関数呼び出しの深度が大きくなるとカーネルメモリが不足することが予想される．そこで，今回の実装方式としてすべての制御データをハッシュ化することで 1 回の退避処理で保護用スタックに積み上げるデータサイズを一定にし，カーネルメモリの使用量を減らすことを提案する（次章で詳述）．

制御データの検査処理

制御データの検査処理は，保護用スタックへ退避させた制御データとコールスタックに積まれている制御データがすべて一致するかを検査する．この検査によって，スタック偽装攻撃によるコールスタックの書き換えを検知でき，プログラムを停止させる．

制御データの検査処理は，関数呼び出し元への復帰処理の直前に行う．復帰処理時の直前に検査を行うことで，関数内で発生したスタック偽装攻撃を，攻撃者の意図する復帰が実行される前に，すなわち攻撃を未然に確実に防ぐことができる．

4. 実装

提案手法を実現するために，以下の 3 点を実装した．

- 制御データの退避処理を行うシステムコール
- 制御データの検査処理を行うシステムコール
- システムコールを呼び出すライブラリ関数

4.1 制御データの退避処理を行うシステムコール

制御データの退避処理を行うシステムコールは，関数のフレームポインタの値を引数にとり，制御データの退避を行う．引数に渡すフレームポインタの値は，レジスタ ebp

© 2012 Information Processing Society of Japan

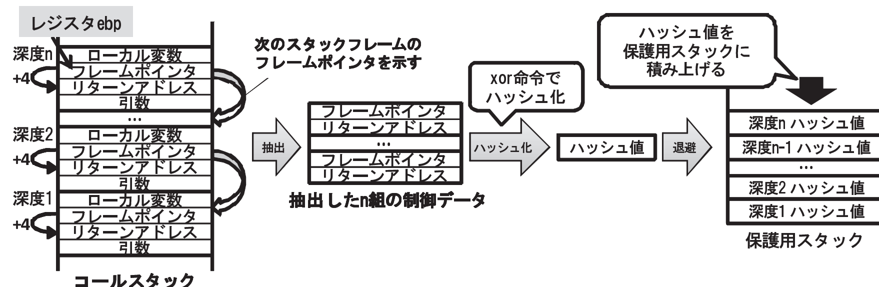

図 6　制御データの退避処理
Fig. 6　Saving process of control data.

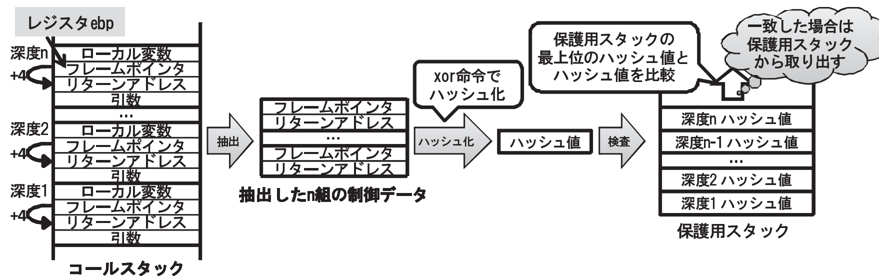

図 7　制御データの検査処理
Fig. 7　Checking process of control data.

から取得する．退避処理は，**図 6** に示す深度 n の関数について以下の 3 つの処理を順に行う．

(1) コールスタックから n 個のリターンアドレスとフレームポインタの値の組を抽出

(2) 上記 (1) で抽出した 2 つの値の n 組すべてを xor 命令でハッシュ化

(3) ハッシュ化した値（1 語）を保護用スタックの最上位に積み上げる

上記の処理において，フレームポインタ値は両システムコールで呼び出されるカーネル関数内で，仮想メモリアドレスから物理メモリアドレスに変換された値を利用している．プログラムの実行ごとに物理メモリアドレスは変化し，ユーザプログラムからは見ることができないため，このハッシュ値まで見越してスタック偽装することはほとんど不可能である．すなわち，この実装方式により，提案手法の攻撃耐性をさらに高めている．メモリ操作にはスラブアロケータを利用し，メモリ使用量の効率化と CPU キャッシュメモリの効率化を図る．この場合，カーネル関数である kmalloc 関数を用いたメモリ確保より高速なメモリ確保が可能である．

4.2　制御データの検査処理を行うシステムコール

制御データの検査処理を行うシステムコールは，関数のフレームポインタの値を引数に渡し，制御データの検査を行う．制御データの検査処理は**図 7** に示す深度 n の関数について以下の 3 つの処理を順に行う．

(1) コールスタックから n 個のリターンアドレスとフレームポインタの値の組を抽出

(2) 抽出したリターンアドレスとフレームポインタの値 n 組を xor 命令でハッシュ化

(3) ハッシュ値が保護用スタックに積み上げられたハッシュ値と一致しているかを検査

上記 (3) の検査の結果，値が一致すれば正常であると判断し，保護用スタックの最上位の値を解放しリターンする．値が一致しない場合は，なんらかの攻撃によってコールスタックが書き換えられたと判断し，当該システムコール内で do_exit 関数を呼び出しプロセスを強制終了させる．

4.3　システムコールを呼び出すライブラリ関数

制御データの退避処理と検査処理を行うシステムコールのそれぞれを呼び出す 2 つの関数 __cyg_profile_func_enter と __cyg_profile_func_exit を新たに共有ライブラリ libsecure.so として今回実装した．この共有ライブラリをプログラムを実行する際にプリロードし使用する．ユーザ関数の入口と出口それぞれに，これら両関数の対応する側を呼び出すコードを挿入することによって，ユーザ関数呼び出しの直後に退避処理，呼び出しから復帰する直前に検査処理を実行する．具体的には，GCC のコンパイルオプションである -finstrument-functions を利用してコンパイルすると，関数の入口と出口それぞれに，これら両関数のうちの対応する側を呼び出す関数呼び出しが挿入される．

これら両関数には，それぞれ制御データの退避を行うシステムコールの呼び出しと制御データの検査を行うシス

© 2012 Information Processing Society of Japan

情報処理学会論文誌 Vol.53 No.9 2075–2085 (Sep. 2012)

テムコールの呼び出しのみを実装する．システムコール呼び出しには int 命令より高速な sysenter 命令を利用する．システムコールに渡す引数には，__cyg_profile_func_enter や __cyg_profile_func_exit のスタックフレームのフレームポインタが指し示す値を渡す．

5. 実験と評価

サンプルプログラムによるスタック偽装攻撃の検知確認とオーバヘッド測定を行った．**表 1** に評価環境を示す．

5.1 サンプルプログラムによるスタック偽装攻撃

2.1 節で述べた代表的な 2 種類の脆弱性（1）バッファオーバフロー脆弱性，および，(2) 書式文字列脆弱性を持つ簡単なサンプルプログラムに対して，スタック偽装攻撃を行い提案手法によって攻撃が検知できることを確認した．

テストするスタック偽装攻撃の事前準備として特定の関数呼び出しが行われた際のコールスタックをダンプしておき，そのダンプ結果を用いてバッファオーバフロー攻撃または書式文字列攻撃によりコールスタックを書き換えた．提案手法を適用せずにこれらのサンプルプログラムを実行した場合，コールスタックの書き換えによって変更された制御フローに基づいた動作が行われ，スタック偽装攻撃の発生を確認した．一方，提案手法を適用してこれらのサンプルプログラムを実行した場合，スタック偽装攻撃による制御フローの変更を検知し，プロセスは強制終了された．

5.2 処理時間に関するオーバヘッド

提案方式による実行時間のオーバヘッドについて述べる．

5.2.1 制御データの退避処理と検査処理にかかるオーバヘッド

関数呼び出しごとに実行される制御データの退避処理と検査処理のシステムコールと getpid システムコールそれぞれの処理時間を測定した．測定結果を**図 8** に示す．getpid システムコールは最も処理が少ないシステムコールである．すなわち，getpid システムコールの処理にかかるオーバヘッドの大半は，ユーザモードからカーネルモード，カーネルモードからユーザモードへとモード変更する際にかかる処理時間である．

したがって，提案手法の両システムコールにおいても，必ずこの getpid の処理時間（約 0.1 マイクロ秒）にほぼ等しい部分は，毎回コンスタントに必要であるといえる．それ以外の前章で述べた退避および検査処理については毎回呼び出された時点での最上位スタックフレームから最下位スタックフレームまでリンクをたどる処理をともなうため，同図中の x 軸に示す深度に比例した処理時間となることが予想できる．同図の結果は，まさにそれを裏打ちするものとなった．

実際のプログラムでのオーバヘッドは，関数の深度と呼び出される回数，頻度に依存する（次項から詳述）．

5.2.2 gzip の時間オーバヘッド

gzip バージョン 1.2.4 に対して提案手法を適用し，gzip 本来の動作の確認とオーバヘッドの測定を行った．gzip のオーバヘッド測定では，Linux Kernel のソースコードのファイルの解凍にかかる時間を測定した．解凍を 1,000 回実行して測定された時間の平均値を**表 2** に示す．実行時に発生した関数呼び出しの総数は 51,843 回である．関数呼び出しの平均深度は 8.421 であり，gzip の実行中に再帰関数は存在しない．入出力処理のオーバヘッドをできるだけ少なくするため，カーネルソースコードは RAM ディスクに置き，解凍先も同じ RAM ディスクを指定した．実行時間は，提案手法を適用しない場合で 5,477 ミリ秒，適用した場合で 5,619 ミリ秒であり，提案手法による増加時間は 142 ミリ秒である．gzip の解凍処理の場合，解凍処理そのものにかかる時間が提案手法によるオーバヘッドと比較して大きいため，増加した処理時間が実行時間全体に占める割合は 2.52%と小さな値になった．

図 8 で平均深度 8.4 をあてはめると制御データの退避処理と検査処理を行うシステムコールの平均処理時間はそれぞれ約 1.3 マイクロ秒と分かる．この値から，提案手法によるオーバヘッド時間は，総関数呼び出し数 51,843 回×平均システムコール処理時間 1.3 マイクロ秒 × 2（退避処理と検査処理）より約 134 ミリ秒と予測できる．実験におけるオーバヘッドは 142 ミリ秒であり，これはすなわち，実験結果とモデルによる解析結果とが示す値がほぼ一致して

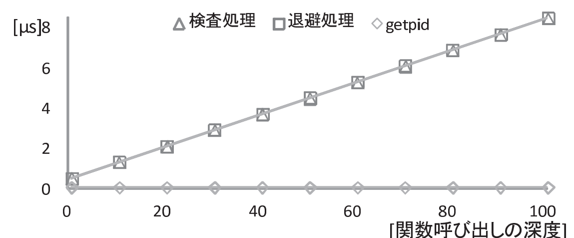

図 8 制御データの退避処理と検査処理にかかる時間
Fig. 8 Saving and checking time.

表 1 評価環境
Table 1 Evaluation environment.

CPU	Intel Core Duo（1.86 GHz）
メモリ	1 GB
ディストリビューション	Fedora 15
Kernel	Linux Kernel 2.6.38

表 2 gzip のオーバヘッド
Table 2 Overhead of gzip process.

		実行時間	増加時間
1.	通常実行	5,477 ms	–
2.	提案手法の適用	5,619 ms	142 ms

© 2012 Information Processing Society of Japan

表 3　httpd のオーバヘッド
Table 3　Overhead of httpd process.

	通常実行	提案手法の適用		
httpd が送信するファイルサイズ	Time per request	Time per request	増加処理時間	処理時間全体に占める割合
4 KB ファイル	1.999 ms	6.916 ms	4.917 ms	71.09%

表 5　wc のオーバヘッド
Table 5　Overhead of wc process.

	通常実行	提案手法の適用		
wc で計測するファイル	実行時間	実行時間	増加処理時間	処理時間全体に占める割合
20 MB ファイル	1,530 ms	29,580 ms	28,050 ms	94.82%

表 4　MPM の設定
Table 4　MPM settings.

設定項目	設定内容
使用する MPM	Prefork
StartServers	0
MaxSpareServers	1
MinSpareServers	1
MaxClients	1
MaxRequestsPerChild	0

いることが分かる．

5.2.3　httpd の時間オーバヘッド

httpd（Apache HTTP Server 2.2.21）に対して提案手法を適用し，本来の動作の確認とオーバヘッドの測定を行った．httpd はデーモンとして動き続けるプログラムであるため，プログラムの開始から終了までの処理時間を求めることは困難である．そこで，ApacheBench [22] を用いて 1 リクエストの処理にかかる時間とオーバヘッドの測定を行った．測定結果を**表 3** に示す．この測定では，httpd が動作する計算機と ApacheBench を実行する計算機に分けて計測を行った．ともにローカルエリア内にある計算機で，ハブ経由で接続されている．4 KB のファイルに対して同時接続数 1 で 10,000 回発行されたリクエストの処理に要する平均時間を測定した．

httpd は複数のリクエストを同時に処理するために MPM（Multi Processing Module）を用いて子プロセスを複数作成する．評価環境にマルチコア CPU を利用したため，複数のプロセスによってリクエストが処理されると正確なオーバヘッド測定が困難である．そのため，動作させる子プロセス数を 1 つに制限した測定時における設定を**表 4** に示す．

処理時間は，提案手法を適用しない場合で 1.999 ミリ秒，適用した場合で 6.916 ミリ秒であり，提案手法による増加した処理時間は 4.917 ミリ秒となった．httpd の処理の場合，1 リクエストに対する処理が 1.999 ミリ秒と小さいため，増加した処理時間が実行時間全体に占める割合は約 71%と大きくなった．gzip のオーバヘッドの増加分 142 ミリ秒と httpd におけるオーバヘッドの増加分 4.917 ミリ秒の比較から httpd における増加した処理時間そのものの意味では比較的小さい値であることが分かる．

5.2.4　wc の時間オーバヘッド

wc（GNU Core Utilities 8.13）に対して提案手法を適用し，本来の動作の確認とオーバヘッドの測定を行った．今回の実験では，次章で述べる先行研究との比較のため，wc の入力として，英文文字列を含む 20 MB のファイルの処理に要する時間を測定した．wc を 1,000 回実行して測定された時間の平均値を**表 5** に示す．1 回の実行で発生した関数呼び出しの総数は 20,759,324 回，関数呼び出しの平均深度は 3.999 であった．入出力処理のオーバヘッドをできるだけ少なくするために，入力ファイルは RAM ディスクに置いた．実行時間は，提案手法を適用しない場合で 1,530 ミリ秒，適用した場合で 29,580 ミリ秒であり，提案手法による増加時間は 28,050 ミリ秒である．wc の処理は，ファイルサイズと総関数呼び出し数がほぼ完全に比例し，ファイルサイズが増加するにつれ関数呼び出し数が多くなる．関数呼び出しの平均深度は 3.999 であり，再帰関数はプログラム中に存在しない．このことは，ファイルサイズには依存しない．この wc では，提案方式の時間オーバヘッドが非常に大きなものとなっているように見える．しかし，これは実際には，wc は本来その文字数を数えるという処理が軽いにもかかわらず，関数呼び出しの総回数が多いため，提案手法のオーバヘッドが際立っているにすぎない．いい換えれば，wc のようなプログラムは，オーバヘッドで見る限り，最悪のケースとなっている．

次に，gzip の場合と同様に，図 8 のデータを用いて，表 5 の結果を簡単に検証する．図 8 において，今回の平均深度 3.999 に相当する提案手法の両システムコールの平均オーバヘッドはそれぞれ約 0.67 マイクロ秒である．この値から，提案手法により増加すると予想されるオーバヘッドの総時間は，総関数呼び出し数 20,759,324 回×平均システムコールオーバヘッド約 0.67 マイクロ秒× 2（退避処理と検

© 2012 Information Processing Society of Japan

査処理）より約 27,817 ミリ秒と予測できる．実験におけるオーバヘッド時間は 28,049 ミリ秒であり，この場合にもモデルによる解析値と実際の値とがほぼ一致しているといえる．

6. 考察

対象となる攻撃の範囲

提案手法の検知対象範囲はスタック偽装攻撃のみを対象としており，提案手法は原理的にスタック偽装攻撃による制御フローの変更を関数ごとに検知することが可能である．さらに，スタック偽装攻撃より単純な通常のコードインジェクション攻撃や Return-into-libc 攻撃を防ぐことも可能である．一方で，2.3 節で述べた既存の攻撃検知システムでどのようなスタック偽装攻撃でも完全に止められるものは存在しない．ただし，阿部ら [18] と槇本ら [19] の手法では，検知可能な場合があることが，論文中で述べられている．検知目的，検知項目，そして，検知タイミングが異なるため，本提案手法と槇本らの手法を組み合わせることにより，攻撃耐性をより高めることが可能である．

時間オーバヘッド

前章で述べたとおり提案手法による処理時間が実行時間全体に占める割合は gzip で 2.52%，httpd で 71.09%，wc で 94.82%となった．ハードウェアや OS を含む実行環境が異なるため，一律に比べることはできないが，文献に記載された他の類似システムと比較してみる．Wagner らのシステム [11] では sendmail で 1 時間以上のオーバヘッド，阿部らのシステム [18] では 22 MB のファイルを用いた wc の測定で 73.68%，槇本らのシステム [19] では 20 MB のファイルを用いた wc の測定で 49.24%がシステムの処理にかかる割合であった．

適用の容易性

本論文で提案している制御データの退避処理と検査処理は，Loadable Kernel Module として実装することが可能であり，多くの計算機に対して手軽に導入できる．さらに，通常の call 命令，return 命令の動作に組み込みハードウェア化することすら可能である．それは，処理時間オーバヘッドを大きく削減できる可能性もあることから，検討に値すると考える．Wagner らなどの静的解析情報を用いた手法では，事前にソースコードの解析を行いセキュリティポリシの作成を管理者が行う必要があり敷居が高いといわざるをえない．一方，提案手法では，オプション付きの GCC で再コンパイルするのみで容易に検査コードをすべての関数の入り口と出口に対して埋め込むことが可能であるため，容易に導入することが可能である．

ハッシュ値の衝突

提案手法では，攻撃者により制御データの退避処理によって退避されたハッシュ値と一致するようにコールスタックを書き換えられた場合，制御データの検査処理で検知できないという問題がある．しかし，4.1 節，4.2 節で述べたとおり，ハッシュ値の生成にはプログラムの実行ごとに変化する物理メモリアドレスを使用しているため，攻撃者が退避されたハッシュ値と一致するようにコールスタックを書き換えることは困難である．また，xor 命令以外のハッシュアルゴリズムを用いると強度をさらに高めることが可能であるが，時間オーバヘッドとのトレードオフとなる．同様に，プログラム開始時のクロックを毎回のハッシュ計算で使えば，攻撃者が故意に一致させることは，さらに困難となる．しかし，どうしても，偶然一致する見逃しは避けられない．

制御フローを退避させるシステムコールの悪用

スタック偽装攻撃によって制御フローが偽装された後に制御データを退避させるシステムコールが発行された場合，攻撃が行われた関数内で制御フローの書き換えを検知できなくなる．しかし，攻撃者が当該のシステムコールを発行するためには，コード埋め込みなどを行い制御データを書き換えなければならない．すると，その段階で提案手法はその制御データの書き換えを検知できるため，既存の攻撃を用いて当該システムコールを任意の場所から発行することはできない．

7. おわりに

本論文では，多くの侵入検知システムにおいて検知できないスタック偽装攻撃を検知するための手法を提案した．提案手法では，関数呼び出し時にコールスタックに積まれたすべてのフレームポインタとリターンアドレスからなる制御データをカーネル空間に存在する保護用スタックに退避させる．そして，関数呼び出し元への復帰処理時に保護用スタックに退避させた値を用いてコールスタックに積まれている制御データを検査する．提案手法を実現するために，新たなシステムコールの追加を行い，GCC を用いて関数呼び出し時と復帰処理時に作成したシステムコールを呼び出す処理を挿入する．提案手法を適用する際のオーバヘッドを削減するためにシステムコールの呼び出しに sysenter 命令を利用した．カーネル空間でのメモリ取得にはスラブアロケータを利用し，高速なメモリ管理を実現した．提案手法を用いることによって，従来の侵入検知システムではほとんど検知できなかったスタック偽装攻撃が検知できることを確認した．gzip と httpd および wc において提案手法を適用したときの実行時間を測定し，オーバヘッドが処理時間の全体に占める割合は gzip で 2.52%，httpd で 71.09%，wc で 94.82%であった．提案手法では，オーバヘッドを他の先行研究と同程度に抑えつつ，他の先行研究では防ぐことが困難なスタック偽装攻撃を防ぐことができる．今後の課題として，関数ポインタやローカル変数の書き換えによる不正な関数呼び出しとオーバヘッドの削減の 2 点があげられる．

© 2012 Information Processing Society of Japan

参考文献

[1] Forrest, S., Hofmeyr, S., Somayaji, A. and Longstaff, T.: A Sense of Self for Unix Processes, *Proc. IEEE Symposium on Security and Privacy* (*SP '96*), pp.120–128, IEEE Computer Society (1996).

[2] Wagner, D. and Soto, P.: Mimicry attacks on host-based intrusion detection systems, *Proc. ACM Conference on Computer and Communications Security* (*CCS '02*), pp.255–264, ACM (2002).

[3] Kruegel, C., Kirda, E., Mutz, D., Robertson, W. and Vigna, G.: Automating mimicry attacks using static binary analysis, *Proc. 14th Conference on USENIX Security Symposium* (*SSYM '05*), pp.11–26, USENIX Association (2005).

[4] Yves, Y.: An overview of common programming security vulnerabilities and possible solutions, Master's thesis, Vrije Universiteit Brussel (2003).

[5] Shankar, U., Talwar, K., Foster, J. and Wagner, D.: Detecting format string vulnerabilities with type qualifiers, *Proc. 10th Conference on USENIX Security Symposium* (*SSYM '01*), pp.16–31, USENIX Association (2001).

[6] Newsham, T.: Format String Attacks (2001), available from ⟨http://hackerproof.org/technotes/format/formatstring.pdf⟩.

[7] Molnar, I.: Exec-shield, available from ⟨http://people.redhat.com/mingo/exec-shield/⟩.

[8] The PaX Team: The pax project, available from ⟨http://pax.grsecurity.net/⟩.

[9] Cown, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q. and Hinton, H.: Stack-Guard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, *Proc. 7th Conference on USENIX Security Symposium* (*SSYM '98*), pp.63–78, USENIX Association (1998).

[10] 江藤博明，依田邦和：propolice：スタックスマッシング攻撃検出手法の改良，情報処理学会論文誌，Vol.43, No.12, pp.4034–4041 (2002).

[11] Wagner, D. and Dean, D.: Intrusion Detection via Static Analysis, *Proc. IEEE Symposium on Security and Privacy* (*SP '01*), pp.156–168, IEEE Computer Society (2001).

[12] Feng, H., Giffin, J., Huang, Y., Jha, S., Lee, W. and Miller, B.: Formalizing Sensitivity in Static Analysis for Intrusion Detection, *Proc. Symposium on Security and Privacy* (*SP '04*), pp.194–208, IEEE Computer Society (2004).

[13] Giffin, J., Jha, S. and Miller, B.: Efficient context-sensitive intrusion detection, *Proc. Network and Distributed System Security Symposium* (*NDSS*) (2004).

[14] Zhang, X., Li, J., Jiang, Z. and Feng, H.: Black-Box Extraction of Functional Structures from System Call Traces for Intrusion Detection, *ICIC* (*3*), Communications in Computer and Information Science, Vol.2, pp.135–144, Springer (2007).

[15] Sekar, R., Bendre, M., Dhurjati, D. and Bollineni, P.: A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors, *Proc. IEEE Symposium on Security and Privacy* (*SP '01*), pp.144–155, IEEE Computer Society (2001).

[16] Feng, H., Kolesnikov, O., Fogla, P., Lee, W. and Gong, W.: Anomaly Detection Using Call Stack Information, *Proc. IEEE Symposium on Security and Privacy* (*SP '03*), pp.62–75, IEEE Computer Society (2003).

[17] Gao, D., Reiter, M. and Song, D.: Gray-box extraction of execution graphs for anomaly detection, *Proc. ACM Conference on Computer and Communications Security* (*CCS '04*), pp.318–329, ACM (2004).

[18] 阿部洋丈，大山恵弘，岡　瑞起，加藤和彦：静的解析に基づく侵入検知システムの最適化，情報処理学会論文誌：コンピューティングシステム，Vol.45, No.3, pp.11–20 (2004).

[19] 槇本裕司，齋藤彰一，古屋雄介，白井宏憲，上原哲太郎，松尾啓志：ライブラリ関数呼び出し監視による侵入防止システムの実現，情報処理学会論文誌：コンピューティングシステム，Vol.3, No.1, pp.38–49 (2010).

[20] 服部眞也，毛野高彦，桑原寛明，國枝義敏：静的解析情報を利用したセキュアシステムの侵入検知精度向上，情報処理学会第 71 回全国大会論文集，Vol.1, pp.69–70 (2009).

[21] Brandon, B.: Compiler Security Checks In Depth (2002), available from ⟨http://msdn.microsoft.com/en-us/library/Aa290051⟩.

[22] ApacheBench: A complete benchmarking and regression testing suite, available from ⟨http://httpd.apache.org/⟩.

冨永 悠生　（正会員）

1987 年生．2010 年立命館大学情報理工学部情報システム学科卒業，2012 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 NTT コムウェア（株）入社，現在に至る．ホスト型侵入検知システム，コンパイラ，セキュア OS 等に興味を持つ．

樫山 武浩　（正会員）

1983 年生．2005 年関西大学総合情報学部卒業，2007 年同大学大学院総合情報学研究科博士課程前期課程修了，2010 年同研究科博士課程後期課程修了，同年関西大学総合情報学部ポストドクトラルフェロー，立命館大学立命館グローバル・イノベーション研究機構ポストドクトラルフェローとなり，現在に至る．オペレーティングシステム，コンピュータセキュリティ等の研究に従事．また，2004～2010 年（株）関西総合情報研究所．CAD/CG，GIS，画像処理等の研究業務に従事．博士（情報学）．土木学会会員．

© 2012 Information Processing Society of Japan

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．

桑原 寛明　（正会員）

2001年名古屋大学工学部電気電子情報工学科卒業．2003年同大学大学院工学研究科計算理工学専攻博士前期課程修了．同大学院情報科学研究科情報システム学専攻博士後期課程へ進学．2007年立命館大学情報理工学部助教．2009年より同講師．博士（情報科学）．プロセス代数，型理論，プログラム解析等の研究に従事．日本ソフトウェア科学会会員．

毛利 公一　（正会員）

1972年生．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年同大情報理工学部情報システム学科講師，2008年同准教授となり，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，日本ソフトウェア科学会，ACM，IEEE-CS，USENIX各会員．

齋藤 彰一　（正会員）

1993年立命館大学理工学部情報工学科卒業．1995年同大学大学院博士前期課程修了．1998年同大学院博士後期課程単位習得中退．同年和歌山大学システム工学部情報通信システム学科助手．2003年同講師，2005年同助教授．2006年名古屋工業大学大学院助教授，2007年同准教授．現在に至る．オペレーティングシステム，インターネット，セキュリティ等の研究に従事．博士（工学），ACM，IEEE-CS各会員．

上原 哲太郎　（正会員）

1990年京都大学工学部情報工学科卒業．1995年同大学大学院工学研究科情報工学専攻博士後期課程研究指導認定退学．同年同研究科助手，1996年和歌山大学システム工学部講師，2003年京都大学工学研究科附属情報センター助教授，2006年同大学学術情報メディアセンター助教授，2007年同准教授．2011年総務省入省．2002年NPO情報セキュリティ研究所設立に参画，副代表理事．2011年より研究員．京都大学博士（工学）．IEEE，電気学会，電子情報通信学会，日本ソフトウェア科学会，システム制御情報学会，情報ネットワーク法学会，CIEC，コンテンツ文化史学会各会員．

國枝 義敏　（正会員）

1982年京都大学大学院工学研究科修士課程情報工学専攻修了．1982年京都大学工学部情報工学科助手．1991年同助教授．1996年和歌山大学システム工学部教授．2004年立命館大学情報理工学部教授，現在に至る．京都大学工学博士．高性能計算，ディペンダブルシステム等の研究に従事．ACM，IEEE CS，電子情報通信学会各会員．

© 2012 Information Processing Society of Japan