異常検知システムにおける正常動作データのモジュール化

大山恵弘[†] 王 維[††] 加藤和彦[††,†††]

アプリケーションが発行するシステムコールのパターンの学習に基づく異常検知システムが近年注目されている．本稿では，その種類の異常検知システムにおける，正常な動作を表現するデータ（正常動作データ）の作り方の改良法を提案する．既存の方法は正常動作データをアプリケーションごとに作るが，提案する方法はアプリケーションを構成するコンポーネント（たとえば共有ライブラリ）ごとに正常動作データを作る．提案する方法の主な利点は，異常の発生している場所をより高い精度で把握できることと，コンポーネントの正常動作データを複数アプリケーション間で共有および再利用できることである．我々はその方法に基づく異常検知システムを実装し，httpd，ftpd，sshd，Emacs を監視しながら実行する実験を通じて既存のシステムとの比較を行った．

Modularizing Normal Behavior Databases in Anomaly Detection Systems

Yoshihiro Oyama,[†] Wang Wei[††] and Kazuhiko Kato[††,†††]

In recent years, much attention has been paid to anomaly detection systems in which normal behavior of applications is modeled on system call traces recorded in their executions. In this paper, we propose an improved scheme for representing normal behavior databases in anomaly detection systems. While existing schemes create a profile of normal behavior for each *application*, our scheme creates a profile for each *program component* that makes up an application (e.g., it creates a profile for each shared library). The advantages of our scheme are that it can pinpoint the program part causing abnormal behavior and that normal behavior databases of components can be shared and reused among multiple applications. We implemented an anomaly detection system based on the scheme and compared it with an existing system through experiments in which the systems monitored the execution of httpd, ftpd, sshd, and Emacs.

1. はじめに

悪意を持つ者によるバッファオーバフロー等の攻撃やソフトウェアの品質の低さに起因する計算機システムの異常動作が深刻な問題となっている．そこで，計算機システムの異常を早期に発見し対策を講じるための手段として，異常検知システム（anomaly detection system）が盛んに研究されている．中でも，アプリケーションが過去に発行したシステムコールのパターンの学習に基づく技術の研究は近年良い成果をあげている[1),7),10)~14),18),20)]．それらの技術は過去に発行されたシステムコールの記録をもとにアプリケーションの正常な動作をモデル化し，モデルに合わない実行を異常と見なす．その技術を利用した既存の異常検知システムとしては，Forrest らの計算機免疫系[7)]や，Sekar らの有限状態オートマトンを利用した異常検知システム[18)]がある．両システムは**訓練モード**においてシステムコール等の情報を記録しながらアプリケーションを実行する．そして記録をもとにそのアプリケーションの正常な動作を表現する規則（**正常動作データ**）を作る．一方，**検査モード**において，正常動作データとの照合によって異常を検査しながらアプリケーションを実行する．この種類の異常検知システムは未知の攻撃を検知できるという長所を持つ．

本稿では異常検知システムの正常動作データの改良された表現法を提案する．我々は既存技術[1),7),8),10)~14),18)]の，正常動作データをアプリケーションごとに作るという点を見直し，正常動作データをコンポーネントごとに作る．コンポーネントとは独立した再利用可能なプログラムモジュール（ライブラ

† 東京大学
The University of Tokyo
†† 筑波大学
University of Tsukuba
††† 科学技術振興事業団
Japan Science and Technology Corporation

リやプラグインプログラム）を指す．提案する方法では，各コンポーネントの正常な動作を表現する規則を組み合わせて，アプリケーション全体の動作が正常かどうかを判断する．たとえば，モジュール mod_perl が組み込まれた Apache HTTP サーバの異常検知では，httpd バイナリ，mod_perl，libc 等の正常動作データを組み合わせる．すなわち，「Apache 全体としての正常な動作」を定義する代わりに，「httpd バイナリの正常な動作」，「mod_perl の正常な動作」，「libc の正常な動作」等を定義する．コンポーネントの正常動作データは複数のアプリケーション間で共有される．たとえば HTTP サーバの異常検知と FTP サーバの異常検知は libc の正常動作データを共有する．アプリケーションの過去の動作から自動的に各コンポーネントの正常動作データを構築し，かつ，複数のコンポーネントの正常動作データを組み合わせて 1 つのアプリケーションの異常検知を行うシステムは今までに存在しない．

本研究の目的は，コンポーネントを組み合わせて構築されるソフトウェアに適した異常検知システムの構築法の確立である．本研究では上記の方法に基づく異常検知システム SQUIDS を実装し，実験を通じて有効性を評価した．本研究の背景にある考えは，

> アプリケーションは作者も信頼性も用途も異なる独立なコンポーネントの集まりとして構築されている．よって，実行が正常か異常かは，アプリケーション全体としての動作から判断するよりも，各コンポーネント内部の動作とコンポーネントの境界をまたぐ動作から判断するほうが自然である．

というものである．アプリケーション全体としての動作は結合されるコンポーネントに応じて変化する．たとえば Apache サーバが発行するシステムコール列は httpd バイナリにすべてを決められているわけではなく，mod_perl や libc 等のコンポーネントが独立に自分の振舞いを行う結果決まっている．

SQUIDS は 2 つの用途に役立つ．第 1 の用途は，バッファオーバフロー等によりプログラムの制御の流れを不正に変える攻撃の検知である．第 2 の用途は，訓練モードで実行されていないプログラムパスの通過の検知である．一方，システムコール引数等のデータの中身を改変する攻撃，race condition を利用する攻撃[2]，DoS 攻撃の検知には適さない．

SQUIDS は以下の特長を持っている．

- 異常の原因がどのコンポーネントにあるのかをユーザに示す．既存のシステムは正常動作データに合わないシステムコールが呼び出されたことを警告するだけである．その結果，既存のシステムではたとえばアプリケーション側で異常が発生したのか，ライブラリ内部で異常が発生したのかをユーザは把握できない．
- 正常動作データの再利用性が高い．既存のシステムではライブラリ等のコンポーネントを交換したら，各アプリケーションの正常動作データを一から作り直す必要がある．一方 SQUIDS では交換したコンポーネントの正常動作データだけを入れ替えればよく，残りのコンポーネントについては古い正常動作データを使い続けられる．文献 19) のアプローチのように，ベンダーがコンポーネントとその正常動作データをセットにして配布すれば，ユーザは新しいコンポーネントを導入した後に自分で正常動作データを作る必要がない．

本稿の構成は次のとおりである．2 章で異常検知システム SQUIDS を説明する．3 章で実験結果を示す．4 章で本研究を関連研究と比較する．5 章で結論と今後の課題を述べる．

2. 異常検知システム SQUIDS

2.1 データ構造

SQUIDS は正常動作データをオートマトンで表現する．オートマトンの例を**図 1** に示す．各コンポーネント（アプリケーションプログラム自身も含む）の正常な動作は，別コンポーネントへの呼び出しの遷移を示すオートマトンで表現される．現在の実装では動的リンクされる共有ライブラリをコンポーネントとして扱っている．共有ライブラリの正常な動作を表現するオートマトンの端点はシステムコール呼び出しである．アプリケーションの正常動作を表現するオートマトンの端点は，共有ライブラリの関数を呼び出すプログラムポイントである（図 1 では見やすさのために呼び出される関数名も付した）．アプリケーションの正常動作データをライブラリ関数の種別ではなくプログラムポイントで表現する理由は，同じ関数への異なる呼び出し場所を区別することにより，分岐やループ等のプログラムの構造を忠実に正常動作データに反映させるためである[18]．

2.2 訓練モードでの処理

訓練モードでの処理を説明する．SQUIDS はアプリケーションをシステムコール呼び出しで停止させながら実行する．停止させたら，システムコールの種別（番号）を調べた後，アプリケーションの実行スタックを読んでシステムコールを発行した共有ライブラリ関数を調べる．たとえば，「write システムコールが

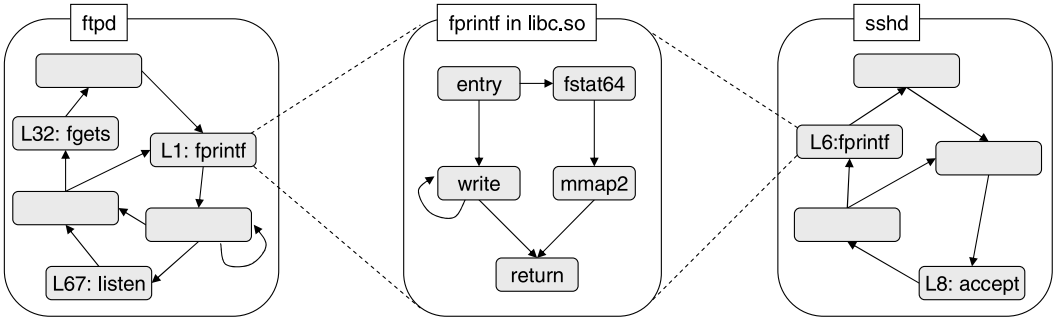


図 1 SQUIDS が作るオートマトンの例
Fig. 1 Example of automaton created by SQUIDS.

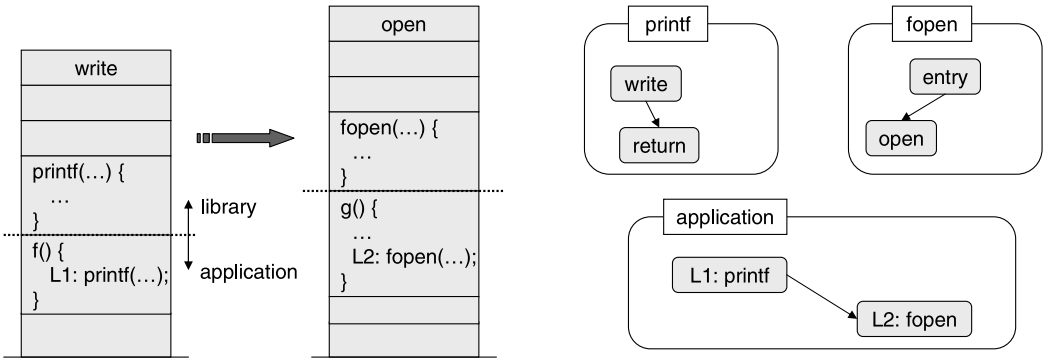


図 2 スタックの変化とその変化によって追加される枝
Fig. 2 Change in stack and edges added due to the change.

libc.so の printf によって発行された」等の情報を得る．次にアプリケーションコード中のプログラムポイントをスタックから得る．たとえば，「その printf の呼び出し元のプログラムポイントは 0x0804b8a5 である」等の情報を得る．ライブラリ関数のオートマトンにシステムコールの遷移を示す枝を加え，アプリケーションのオートマトンにプログラムポイント間の遷移を示す枝を加えた後，アプリケーションの実行を再開する．

連続する 2 つのシステムコールは 1 つのライブラリ関数から呼び出されている場合もあれば異なるライブラリ関数から呼び出されている場合もある．前者の場合にはライブラリ関数のオートマトンに枝が追加される．後者の場合には，**図 2** のように，2 つのライブラリ関数とアプリケーションの合計 3 つのオートマトンに枝が追加される．

2.3 検査モードでの処理

検査モードでの処理を説明する．SQUIDS はアプリケーションをシステムコール呼び出しで停止させながら実行する．停止させたら，システムコールの種別，共有ライブラリ関数の種別，アプリケーションコード中のプログラムポイントをアプリケーションの実行スタックから取得し，正常動作データと照合する．正常動作データのオートマトンが受理しない動作は異常と見なして警報を出す．もしスタックが壊れていたら，バッファオーバフロー等のきわめて危険な異常が発生したとみなし，アプリケーションを即座に終了させる．

図 3 は httpd の異常検知で実際に出された警報の抜粋である．前者は共有ライブラリ関数（apr_dir_read）内のシステムコール呼び出しの遷移の異常を警告している．後者はアプリケーション（httpd）のコード内でのプログラムポイントの遷移の異常を警告している．異常が観測されたコンポーネントと異常な遷移がこのように明示されることは，異常を詳細に調査したり異常への対策を取る上で有用である．

SQUIDS は，過去の実行の学習に基づく多くの異常検知システムと同じく，学習されていない動作をすべて異常と判断する．この性質は，無害な動作にも警報が出る等の不利な点ももたらすが，積極的に活用することによって長所にもなりうる．同様の主張は文献 7) にもあるが，学習されていない動作が観測されることは，アプリケーションがきわめて稀な状態に入ったこ

```
...
WARNING (lib): /usr/local/lib/libapr.so.0.0.0:apr_dir_read[lstat64 -> getdents64]
...
WARNING (app): 0x807aa80: /usr/local/lib/libapr.so.0.0.0:apr_signal -> 0x807aa88: /lib/i686/libc-2.2.4.so:getpid
...
```

図 3 警報の例
Fig. 3 Example of alarms.

とを意味する．それを通知してユーザの注意を喚起することにより，ユーザは潜在的な危険に対して早期に対策を取ることができる．

2.4 システムの詳細

SQUIDS は Linux 上に実装されている．SQUIDS は実行の最初でプロセスを fork する．子プロセスがアプリケーションを実行し，親プロセスが子プロセスを監視して異常検知を行う．親プロセスは子プロセスがシステムコールで停止するように処理をしてから子プロセスを開始させる．その処理は，我々が実装したカーネルモジュールによって行われる．そのカーネルモジュールは，指定したシステムコールでプロセスを停止させる機能，停止したプロセスのシステムコール引数やアドレス空間を読む機能，プロセスを再開させる機能を提供する．アプリケーションを実行しているプロセスが fork を呼び出したら，そのプロセスを監視しているプロセスも fork を呼び出し，派生するすべての子プロセスを別のプロセスが監視する．

ライブラリ関数の情報をスタックから得る処理では，スタック内の各フレームに保存されたプログラムカウンタ（PC）を読む．そして，PC がアプリケーションコード中にあるのか共有ライブラリ関数中にあるのかを調べ，共有ライブラリ関数の呼び出し境界を挟む 2 つのフレームを見つける．境界の上側のフレームから取得した PC を含むライブラリ関数の名前を，共有ライブラリファイルの情報とメモリマップの情報から得る☆．境界の下側のフレームからはアプリケーションコード中の PC を得る．PC が共有ライブラリ関数中のものであるか否かは，Linux の procfs（プロセス・ファイルシステム）を利用して調べる．procfs は，どの共有ライブラリファイルがどの範囲のアドレスにメモリマップされているかの情報を提供する．

アプリケーションが fork する子プロセスの動作は，fork のプログラムポイントごとに作られるオートマトン（つまり親プロセスとは別のオートマトン）を使って記録・検査される．親プロセスの動作は fork 実行前と同じオートマトンを使って記録・検査される．現在の実装はマルチスレッドプログラムには対応していない．アプリケーションが exec を呼び出したら，以降に使われる正常動作データは，exec されたプログラムのそれに切り替えられる．文献 21) の方法と同じく異常検知に貢献しないという理由で，SQUIDS はアプリケーションが発行するメモリ確保システムコール brk を無視する．

2.5 システムの特性

SQUIDS はシステムコールを発行しないライブラリ関数の呼び出しに関する動作を記録・検査しない．このアプローチを取った第 1 の理由は，安全性等の面で通常問題になるのは，システムコールを発行するライブラリ関数のみであるという観測からである．システムコールを発行しないライブラリ関数（たとえば数学関数）の異常のみによって安全性が失われることは稀である．第 2 の理由は，すべてのライブラリ関数の呼び出しを記録・検査するとオーバヘッドが著しく増加するからである．

現在の SQUIDS では静的リンクされたライブラリはアプリケーションと同じコンポーネントに属する，つまり，アプリケーションコードの一部と見なす．しかし，本研究で提案している異常検知の方法自体は，コンポーネントの境界として共有ライブラリの境界以外のものを使うことを妨げない．本研究の方法が要求するものは，システムコール呼び出し時にアプリケーションがコンポーネント境界をどうまたいでいるかの情報を獲得できることである．その要求が満たされるならば，様々な境界をコンポーネントの境界として利用できる．たとえば SQUIDS に若干の改造を加えれば，各関数が属する論理的なコンポーネントをユーザが指示できるような異常検知システムが作れる．脆弱性がありそうな関数群を他の関数とは別のコンポーネントの関数として扱う指示をそのシステムに与えれば，効果的な異常検知が実現できる．

☆ 現在の実装では nm コマンドの出力をもとに各ライブラリ関数のコードのアドレス範囲を割り出している．nm コマンドは共有ライブラリに含まれるシンボルの名前と値を出力する機能を持つ．

3. 実　　験

3.1 実験の設定と位置付け

SQUIDS と文献 18) の方法（Sekar らの方法）に基づく異常検知システムを実装し，実験によって比較した．両システムの大半のコードは共有されており，正常動作データの表現法に依存する部分のコードだけが異なる．Sekar らの方法では 1 つのアプリケーションに対して，ライブラリ関数内部の動作の情報も含む 1 つの巨大なオートマトンが作られる．オートマトンの端点はシステムコールの種別とアプリケーションコード中のプログラムポイントの組である．実験環境は x86 上の Red Hat Linux 7.1（kernel 2.4.7，glibc 2.2.4）である．アプリケーションには Apache httpd 2.0.39，ProFTPD（ftpd）1.2.5，OpenSSH 3.4 sshd を使用した．httpd はユーザ権限で，ftpd と sshd は root 権限で実行した．SQUIDS には，検査モードで異常を警告した後にその異常をすぐ正常動作データに加え，同じ警報を繰り返し出さない実行オプションが用意されている．すべての実験でそのオプションを使用した．実験では訓練モードで各アプリケーションの正常動作データを作るための**訓練コマンド列**を使用した．訓練コマンド列は付録に掲載する．

実験の目的の 1 つは，現実的なアプリケーションの異常検知において，正常動作データのサイズ，誤警報（false alarm）の数，異常の検知率，アプリケーションの性能に与える影響等の点で，本研究で提案する方法と既存の方法がどれくらい異なるかを定量的に知ることである．もう 1 つの目的は，いくつかのアプリケーションを実行して得たコンポーネントの正常動作データを別のアプリケーションの異常検知で再利用することがどの程度現実的かを調べることである．

3.2 正常動作データの増加の様子

訓練コマンド列の実行によって正常動作データが増加する様子を測定した．アプリケーションが呼び出したシステムコールの数と正常動作データのオートマトンのサイズの関係を**図 4**，**図 5**，**図 6** に示す．オートマトンのサイズとして枝の数を利用した．図中の monolithic approach は Sekar らの方法，our approach は本研究で提案する方法を示す．libapr は Apache に付属する共有ライブラリであり，他のアプリケーションでは使用されない．どのアプリケーションでもライブラリ関数のオートマトンは早く収束した．訓練モードの初期段階で急激にサイズが増加した後，ほとんど増加しなくなる．一方，アプリケーションのオートマトンは実行全体を通じて増加を続ける傾向がみられた．

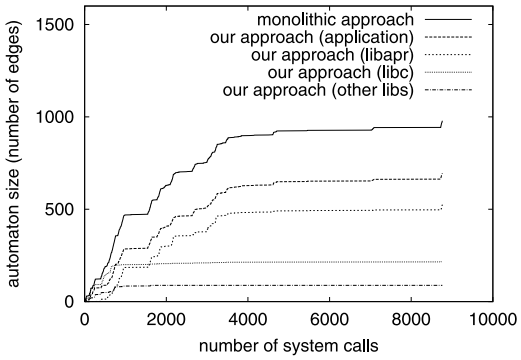


図 4 httpd の正常動作データが増加する様子
Fig. 4 Increase in normal behavior database of httpd.

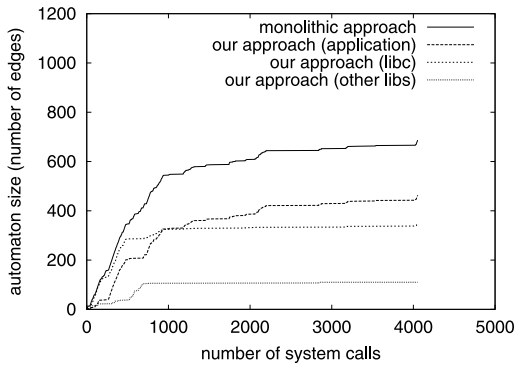


図 5 ftpd の正常動作データが増加する様子
Fig. 5 Increase in normal behavior database of ftpd.

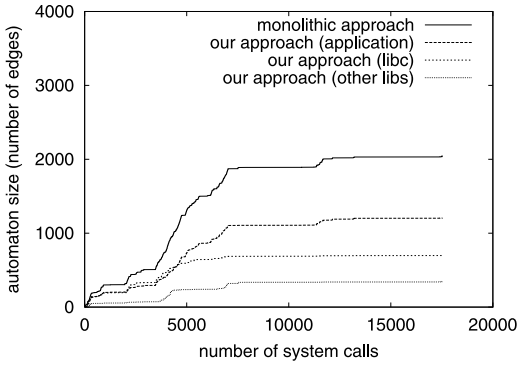


図 6 sshd の正常動作データが増加する様子
Fig. 6 Increase in normal behavior database of sshd.

ただし，どのアプリケーションでも実行の前半ではサイズが急激に増加するもののすぐに増加が鈍化し，実行の後半では増加幅は非常に小さいものになる．

3.3 正常動作データのサイズ

httpd，ftpd，sshd の訓練コマンド列をすべて実行して得られる正常動作データのサイズを**図 7** に示す．本研究の方法では Sekar らの方法よりも各アプリケー

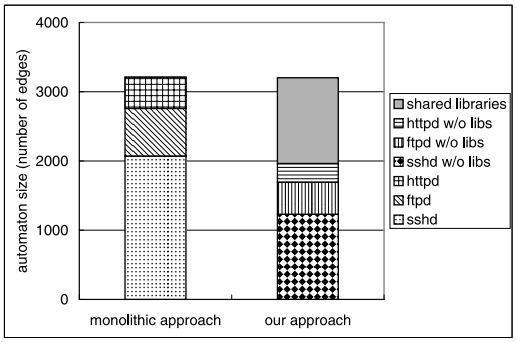


図 7 正常動作データのオートマトンのサイズ
Fig. 7 Sizes of automata in normal behavior database.

ションのオートマトンのサイズが小さくなり，共有ライブラリのオートマトンも含む正常動作データのサイズの総和もわずかに小さくなった．

正常動作データのサイズの総和は本研究の方法によって減少することも増加することもある．1 回だけしかシステムコールを呼び出さない関数の動作をアプリケーションと別のオートマトンに移すと，アプリケーションのオートマトンの枝と端点が減らないまま，新しいオートマトンの分の枝と端点が増える．その結果サイズの総和が増加する．逆に，システムコールを多数回呼び出す関数の動作を別のオートマトンに移すと，アプリケーションのオートマトンの枝が多く減る場合にサイズの総和が減少する．

3.4 誤 警 報

訓練コマンド列で httpd の正常動作データを作り，そのデータで異常検知をしながら httpd を実行した．その httpd に Netscape ブラウザから様々な要求を送ったときに出される誤警報の数を測定した．ブラウザ上ではリンクのクリック，URL の直接入力，強制リロード等の様々な操作を行った．要求の処理のために httpd はシステムコールを 5,267 回実行した．その間に出された誤警報の数を**表 1** に示す．本研究の方法では，アプリケーションの動作に対し 20 の誤警報が，Apache 付属のライブラリ libapr の動作に対し 13 の誤警報が出された．Sekar らの方法では，本研究の方法で出された誤警報の総和よりも 1 個多い 34 個の誤警報が出された．

本研究の方法では Sekar らの方法よりも誤警報は減少する．その理由を**図 8**，**図 9** を用いて説明する．一般にライブラリ関数の動作は引数やプロセスの内部状態の影響を受ける．同じプログラムポイントから同じライブラリ関数を呼び出しても，異なるシステムコールが発行される場合がある．たとえば，fclose 関数はストリームのバッファリングの状態に応じて，図 9 のように write システムコールを発行しない動作（パス A）と発行する動作（パス B）の 2 通りの動作を示す☆．図 8 のプログラムの訓練モードでの実行において，L3 からの fclose の呼び出しではパス A だけが実行されたとする．そのとき，Sekar らの方法では図 8 のオートマトンが作られる．そのオートマトンは L3 から write システムコールを呼び出す動作を含まないため，検査モードで L3 からの fclose の呼び出しがパス B を通ると誤警報が出る．一方，本研究の方法では様々な場所から呼び出された fclose の動作が 1 つのオートマトンに融合される．少なくとも 1 つのアプリケーションの訓練モードでの実行で 1 回でもパス B を通れば，図 9 のような，パス A とパス B の両方を

表 1 httpd の異常検知で出された誤警報の数
Table 1 Number of false alarms raised in anomaly detection of httpd.

Sekar らの方法	34
本研究の方法（アプリケーション）	20
本研究の方法（libapr）	13
本研究の方法（他のライブラリ）	0

```
f()
{
    L1: fopen();
    L2: ...
    L3: fclose();
    L4: ...
}
```

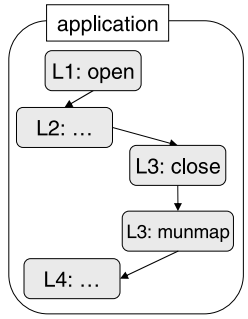


図 8 Sekar らの方法が作るオートマトン
Fig. 8 Sample program and corresponding automaton created in Sekar, et al.'s approach.

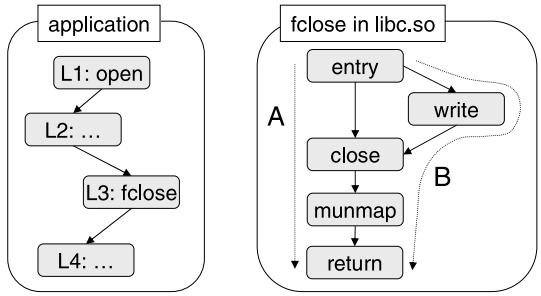


図 9 本研究の方法が作るオートマトン
Fig. 9 Automaton created in our approach.

☆ 実験で得た fclose のオートマトンはさらに多くの枝を持つが，説明を分かりやすくするため，簡単化したオートマトンを示した．

持つオートマトンが作られる．したがって，検査モードで L3 からの fclose の呼び出しがパス B を通っても警報は出ない．

誤警報が少ないことは異常検知システムにとって長所である．しかし，一般に誤警報が少ないシステムは検知率が低いシステムになる傾向があり，誤警報の数と検知率はトレードオフの関係にある．本研究の方法では，各アプリケーションおよび各呼び出し場所にとって関数の正常動作データが過剰に緩いものになり，異常の検知率が下がる可能性がある．たとえば，アプリケーション X はライブラリ関数 f を決まった引数でのみ呼び出し，f は X の実行においては限られた動作しか示さないかもしれない．しかし，もし他のアプリケーションが f を様々な引数で呼び出して f の正常動作データを拡大させると，X における f の実行で本来異常であるものが正常と判定されうる．ユーザは誤警報の数と検知率のトレードオフを理解し，自分の要求に合う性質を持つ異常検知システムを選択する必要がある．

3.5 異常の検知率

異常の検知率の評価を行った．まず訓練モードで httpd，ftpd，sshd の訓練コマンド列を実行して正常動作データを作成した．次に検査モードでそれらのアプリケーションを実行した．ただし検査モードでは人工的に異常を発生させ，異常検知システムがその異常を検知するかどうかを調べた．人工的な異常として，アプリケーションが発行するシステムコール 10 回につき 1 回，システムコールを別のものに変えた効果を作った．システムコールの変化は異常検知システムの内部だけに存在する仮想的な効果であり，アプリケーションは元のシステムコールを実行する．どのシステムコールに変えるかは，それまでの実行での各システムコールの出現確率と同じ確率でランダムに決める．実験の結果を**表 2** に示す．本研究の方法の導入による検知率の変化は httpd と ftpd で 1%未満，sshd で 3%未満の低下にとどまった．

検知率の低下が小さい理由を以下に述べる．本実験で作ったような異常の検知率は，正常動作データの各状態において発行が正常と見なされるシステムコールの個数の平均値（branching factor）[21] と密接に関係している．Branching factor が小さい正常動作データを使用すると，変化させたシステムコールが正常動作データに合わなくなる可能性が高くなるので，高い検知率が得られる．本研究の方法が異常の検知率を大きく低下させていない理由は，スタック内の制御の情報を利用してプログラムの状態を細かく分けることにより branching factor を小さく保っているからである．httpd，ftpd，sshd の正常動作データの branching factor は，Sekar らの方法では 1.16，1.18，1.21 であり，本研究の方法では 1.60，1.64，1.54 であった．一方，広く研究されている N-gram と呼ばれる方法[7] で作られる正常動作データの branching factor は 15 を超えうる[21]．N-gram では正常動作データにシステムコールの種別以外の情報が含まれない．N-gram のような限られた情報を用いる方法は，本研究の方法や Sekar らの方法のような多様な情報を用いる方法に比べ，検知率が低い傾向がある．ただし，システムコールの情報のみを用いる方法の多くには実装が単純でありオーバヘッドが小さいという利点があり，優劣の単純な比較はできない．

表 2 異常の総数，検知数，検知率
Table 2 Number of anomalies, number of detected, and detection rate.

	httpd	ftpd	sshd
Sekar らの方法	847/856	400/403	1645/1696
	(98.9%)	(99.3%)	(97.0%)
本研究の方法	839/856	397/403	1596/1696
	(98.0%)	(98.5%)	(94.1%)

正常動作データの branching factor を元に理論的な異常の検知率を計算できる．上の実験ではシステムコールを変化させる選択肢は 45 個あった．httpd の異常検知において，人工的な異常が正常と判断されて見逃される確率は，Sekar らの方法では $(1.16-1)/45$ である．これは，1,000 回の異常のうち 3 回から 4 回程度を見逃すことに相当する．一方，本研究の方法では，その確率は $(1.60-1)/45$ である．これは，1,000 回の異常のうち 13 回程度を見逃すことに相当する．実験では，どのシステムコールに変化させるかを一様な確率でなく過去のシステムコールの出現確率に従わせているので，見逃された異常の数は理論上の値よりも若干多かった．

3.6 オーバヘッド

異常検知システムが実行を監視する場合と監視しない場合での httpd の性能の変化を調べた．httpd が動いている計算機と同じ計算機から wget -r http://localhost:8080/ を実行し，その完了にかかる時間を計測した．この実験では，同コマンドの実行によって作成した正常動作データを使用して異常検知を行ったため，時間計測中に警報は出ていない．結果を**表 3** に示す．監視によって時間は Sekar らの方法で 13%，本研究の方法で 47%増加した．本研究の方法が与えるオーバヘッドは小さくない．このオーバヘッド

表 3 httpd の実行の監視が web データの収集時間に与える影響
Table 3 Effect of monitoring httpd on the time taken for collecting web data.

異常検知なし	34.0 秒
Sekar らの方法で異常検知	38.3 秒
本研究の方法で異常検知	50.1 秒

表 4 httpd がたどるオートマトンの枝の数と，そのうち ftpd と sshd がたどるオートマトンの枝の数
Table 4 Number of edges followed by httpd and number of edges in them followed by ftpd and httpd.

	libc	libc, libapr 以外のライブラリ
httpd がたどる枝数	168	56
うち ftpd，sshd がたどる枝数	163 (97.0%)	38 (67.9%)

は，本研究の方法に要する処理の量が Sekar らの方法よりも多いことと，SQUIDS の実装が十分に最適化されていないことの両方が原因だと考えている．今後実装を改良してオーバヘッドを縮小することを予定している．

3.7 共有ライブラリの正常動作データの共有

3.7.1 サーバアプリケーション間での共有

異なるアプリケーションが共有ライブラリの正常動作データを共有する度合いを調べた．ftpd と sshd の訓練コマンド列の実行で作られたライブラリのオートマトンが，httpd の訓練コマンド列の実行でたどるライブラリのオートマトンの枝のうち何本を含むかを調べた結果を**表 4** に示す．httpd が辿る libc のオートマトンの枝の数は 168 である．そのうち 163 本は ftpd と sshd の実行で作られた正常動作データに含まれている．httpd がたどる libc の動作の大半（97%）が ftpd および sshd と共有されていることから，ftpd と sshd を実行して作った libc の正常動作データを httpd の異常検知で再利用することは現実的であると考える．非 libc ライブラリのオートマトンに関しては，httpd がたどる枝の数は 56 本であるが，そのうち 38 本は ftpd と sshd の実行で作られた正常動作データに含まれている．

ftpd と sshd の実行で作られたライブラリのオートマトンには，httpd が辿らない枝（httpd の異常検知にとっては余分な枝）も存在する．ftpd と sshd の実行で作られたライブラリのオートマトンは 685 本の枝を含む．そのうち httpd が辿った枝は 163 本であり，辿らなかった枝は 522 本である．後者は httpd の異常検知に貢献しない余分な枝である．SQUIDS では訓練モードでの時間的オーバヘッドは作られるオートマトンの枝数に比例する．検査モードでの時間的オーバヘッドは正常動作データのオートマトンのサイズにはほとんど影響されない．各ライブラリ関数の正常動作データはそれぞれ別のファイルに収められている．余分なライブラリ関数の正常動作データは単に参照されないだけなので性能に影響を与えない．一方，消費ディスク量と消費メモリ量の点での空間的オーバヘッドは余分な枝の存在により増加する．ftpd と sshd で作られたライブラリの正常動作データ（サイズ 685）は，httpd だけを使用して作った正常動作データ（サイズ 168）に比べ，約 4 倍の大きさのディスクとメモリを消費する．

表 5 Emacs がたどるオートマトンの枝の数と，そのうち httpd, ftpd，sshd がたどるオートマトンの枝の数
Table 5 Number of edges followed by Emacs and number of edges in them followed by httpd, ftpd, and sshd.

	libc	X11	その他のライブラリ
Emacs がたどる枝数	234	224	43
うち httpd，ftpd，sshd がたどる枝数	148 (63.2%)	0 (0%)	26 (60.5%)

3.7.2 サーバアプリケーションと非サーバアプリケーションの間での共有

3 つのサーバアプリケーション httpd，ftpd，sshd の実行で作られたライブラリのオートマトンが，非サーバアプリケーションにおけるライブラリの動作をどの程度含んでいるかを調べた．非サーバアプリケーションとして Emacs 20.7.1 を用い，付録に掲載した Emacs 用の訓練コマンド列を実行した．結果を**表 5** に示す．httpd，ftpd，sshd は X ライブラリを呼び出さないので，それらの実行で作られた正常動作データは，Emacs の実行における X ライブラリの異常検知には利用できない．他のライブラリに関しては，Emacs がたどるオートマトンの枝のうち，httpd，ftpd，sshd によって辿られていたものは約 6 割であった．表 4 と表 5 の結果は，サーバアプリケーション間で正常動作データを再利用することに比べ，サーバアプリケーションの実行で作った正常動作データを非サーバアプリケーションで再利用することは難しいことを示唆している．多くの動作を包含する有用な正常動作データを作るためには，訓練モードで様々な系統のアプリケーションを実行し，様々な動作をライブラリに実行させる必要があると考えられる．

4. 関連研究

システムコールの記録と検査に基づく異常検知の方法は多数存在する．文献 7) の N-gram と呼ばれる方法は，決まった長さ N（たとえば $N = 6$）のシステ

ムコール部分列の集合を正常動作データとして使う．どの部分列にも含まれないパターンのシステムコールの呼び出しを異常と判定する．文献 14) では N を可変にする拡張が提案されている．文献 10) の方法は，システムコールの呼び出しの遷移を示すオートマトンを正常動作データとして用いる．文献 1)，13) の方法は，各システムコールの出現頻度を正常動作データとして用いる．文献 11)，12) ではそれぞれデータマイニングと情報理論に基づいて異常を判別する手法を提案している．これらの方法は，発行されたシステムコールの種別以外の情報を利用しない．その結果，正常と異常の分類が雑になり，本研究の方法よりも攻撃を見逃しやすいうえ，正常な実行を偽装する mimicry attack[22] によって検知を回避されやすい．また，これらの方法ではアプリケーションを構成する一部のコンポーネントが入れ替えられたら，そのアプリケーション用の正常動作データを一から作り直す必要がある．本研究の方法では，ベンダーがコンポーネントとその正常動作データをセットにして配布すれば，コンポーネントの入れ替え後にユーザが正常動作データを作り直す必要はない．

本研究の方法は文献 18) の方法を拡張したものである．文献 18) の方法は，ライブラリも含めたアプリケーション全体に対して 1 つのオートマトンを作成する．そのため，どのコンポーネントにどういう異常が発生しているかをユーザが正確に知ることや，あるアプリケーションの実行で作られたコンポーネントの正常動作データを他のアプリケーションの異常検知に再利用することができない．

文献 21) の方法は，ソースコードを解析して各関数ごとに正常な動作を示す規則を作り，それらを組み合わせて，アプリケーションが発行するシステムコールの正常な動作を示す規則を静的に作る．この方法は本研究の方法と異なりソースコードを必要とするうえ，オーバヘッドがきわめて大きい．

SoftwarePot[9),24)]，Janus[6)] は，ファイル等の資源に対して行える操作が制限された実行環境（サンドボックス）を提供する．サンドボックスを提供するシステムは，信頼できないコードから重要な資源を隔離することにより計算機システムを守る．一方，異常検知システムは，信頼できないコードの潜在的に危険な動作を検知することにより計算機システムを守る．サンドボックスを提供するシステムと異常検知システムは相補的に用いることができる．

SQUIDS は，システムコール呼び出しの正しい遷移に関するポリシーを制御の位置に従い動的に切り替えるが，他の種類のポリシーを動的に切り替えるシステムは過去にも存在する．阿部らのシステム[23)] では，資源へのアクセスを許可・禁止するポリシーを任意の関数呼び出し命令の場所で切り替えられる．SubDomain[4)] では，アプリケーションが自ら専用のシステムコールを呼び出して資源アクセスに関するポリシーを切り替える．細粒度保護ドメイン[25)] では，1 つのプロセス内の異なるコンポーネントに制御を移す際，専用のソフトウェア割込みによって，メモリ保護等に関するポリシーを切り替える．

Non-executable stack[17)] は，スタック領域にあるコードを実行できないようにしてバッファオーバフロー攻撃を防ぐ機構である．StackGuard[3)] と propolice[5)] は改造 C コンパイラであり，バッファオーバフロー等によるスタックの改変を検知する処理を含んだコードを生成する．これらの機構は，スタック上のデータの改変を通じた攻撃を防止・検知することはできるが，それ以外の攻撃，たとえば，ヒープ上にある関数ポインタを改変する攻撃を防止・検知できない．SQUIDS はそのような攻撃も検知できる．

Fail-Safe C[16)]，CCured[15)] はメモリ操作の安全性を保証する C 言語処理系であり，スタックやヒープ上のデータを不正に改変する攻撃を防ぐことができる．ただし，そのためにはプログラムを専用のコンパイラでコンパイルする必要がある．よって，Fail-Safe C と CCured は，大半の異常検知システムとは異なり，既存のバイナリコードを安全に実行する目的には利用できない．加えて，SQUIDS は過去に実行されていないプログラムパスの通過を検知する（すなわちアプリケーションが稀な状態に入ったことを検知する）目的に利用できるが，上記の不正なメモリ操作を防止する機構[3),5),15)〜17)] はその目的には利用できない．

5. まとめと今後の課題

アプリケーションを構成するコンポーネントごとに正常動作データを作る異常検知システムを提案した．実験では，オーバヘッドや検知率等の点について既存のシステムとの比較を行うとともに，あるアプリケーションの実行で作成したライブラリの正常動作データが別のアプリケーションの実行におけるライブラリの動作をどの程度含むかを調査した．実験では本研究の方法は正常動作データのサイズ，誤警報の数，異常の検知率に小さな影響しか与えなかったが，既存の方法に比べてオーバヘッドが大きく，オーバヘッドの削減が今後の課題として残る．広い文脈で捉えた本研究の主張は「独立したコンポーネントの集合体の安全な実

行は，独立したセキュリティポリシーの集合体によって実現されるべき」である．セキュリティポリシーの分割により，セキュリティポリシーの作成・維持の手間を軽減したり，異常の発生個所の特定を容易にすることを本研究は狙っている．

今後行うべき仕事を以下に述べる．まず，システムコール引数等の情報を利用して，実行が正常か異常かをより高い精度でより高速に判断する方式を開発したい．次に，日常業務用に稼働しているサーバを用いた実験や，脆弱なプログラムと攻撃コードを用いた実験を行いたい．また，共有ライブラリ境界以外をコンポーネント境界として利用することを試みたい．たとえば，静的ライブラリの境界を利用したり，コードの作者，出所，信頼度に応じて境界を変える等の機能をSQUIDS に組み込んでいきたい．

謝辞 筑波大学の神田勝規氏および査読者から有益なコメントをいただいた．ここに感謝する．

参 考 文 献

1) Asaka, M., Onabuta, T., Inoue, T., Okazawa, S. and Goto, S.: A New Intrusion Detection Method Based on Discriminant Analysis, *IEICE Trans. Info. Syst.*, Vol.E84-D, No.5, pp.570–577 (2001).
2) Bishop, M. and Dilger, M.: Checking for Race Conditions in File Accesses, *Computing Systems*, Vol.9, No.2, pp.131–152 (1996).
3) Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P. and Zhang, Q.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, *Proc. 7th USENIX Security Symp.*, pp.63–78 (1998).
4) Cowan, C., Beattie, S., Kroah-Hartman, G., Pu, C., Wagle, P. and Gligor, V.: SubDomain: Parsimonious Server Security, *Proc. 14th Systems Administration Conf. (LISA 2000)*, pp.355–367 (2000).
5) Etoh, H.: GCC extension for protecting applications from stack-smashing attacks. http://www.trl.ibm.co.jp/projects/security/ssp/
6) Goldberg, I., Wagner, D., Thomas, R. and Brewer, E.A.: A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker, *Proc. 6th USENIX Security Symp.*, pp.1–13 (1996).
7) Hofmeyr, S.A., Forrest, S. and Somayaji, A.: Intrusion Detection using Sequences of System Calls, *Journal of Computer Security*, Vol.6, No.3, pp.151–180 (1998).
8) Jones, A. and Li, S.: Temporal Signatures for Intrusion Detection, *Proc. 17th Annual Computer Security Applications Conf.* (2001).
9) Kato, K. and Oyama, Y.: SoftwarePot: An Encapsulated Transferable File System for Secure Software Circulation, *Software Security—Theories and Systems*, Lecture Notes in Computer Science, Vol.2609, pp.112–132 (2003).
10) Kosoresow, A.P. and Hofmeyr, S.A.: Intrusion Detection via System Call Traces, *IEEE Software*, Vol.14, No.5, pp.35–42 (1997).
11) Lee, W. and Stolfo, S.J.: Data Mining Approaches for Intrusion Detection, *Proc. 7th USENIX Security Symp.*, pp.79–94 (1998).
12) Lee, W. and Xiang, D.: Information-Theoretic Measures for Anomaly Detection, *Proc. 2001 IEEE Symp. on Security and Privacy*, pp.130–143 (2001).
13) Liao, Y. and Vemuri, V.R.: Using Text Categorization Techniques for Intrusion Detection, *Proc. 11th USENIX Security Symp.*, pp.51–59 (2002).
14) Marceau, C.: Characterizing the Behavior of a Program Using Multiple-Length N-grams, *Proc. 2000 Workshop on New Security Paradigms*, pp.101–110 (2000).
15) Necula, G.C., McPeak, S. and Weimer, W.: CCured: Type-Safe Retrofitting of Legacy Code, *Proc. 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pp.128–139 (2002).
16) Oiwa, Y., Sekiguchi, T., Sumii, E. and Yonezawa, A.: Fail-Safe ANSI-C Compiler: An Approach to Making C Programs Secure (Progress Report), *Software Security—Theories and Systems*, Lecture Notes in Computer Science, Vol.2609, pp.112–132 (2003).
17) Openwall Project: Linux kernel patch from the Openwall Project. http://www.openwall.com/linux/
18) Sekar, R., Bendre, M. and Bollineni, P.: A Fast Automaton-Based Method for Detecting Anomolous Program Behaviors, *Proc. 2001 IEEE Symp. on Security and Privacy*, pp.144–155 (2001).
19) Sekar, R., Ramakrishnan, C.R., Ramakrishnan, I.V. and Smolka, S.A.: Model-Carrying Code (MCC): A New Paradigm for Mobile-Code Security, *New Security Paradigms Workshop*, pp.23–30 (2001).
20) Tan, K.M.C. and Maxion, R.A.: “Why 6 ?” Defining the Operational Limits of stide, an Anomaly-Based Intrusion Detector, *Proc. 2002 IEEE Symp. on Security and Privacy*, pp.188–201 (2002).

21) Wagner, D. and Dean, D.: Intrusion Detection via Static Analysis, *Proc. 2001 IEEE Symp. on Security and Privacy*, pp.156–168 (2001).
22) Wagner, D. and Soto, P.: Mimicry Attacks on Host-Based Intrusion Detection Systems, *Proc. 9th ACM Conf. on Computer and Communications Security*, pp.255–264 (2002).
23) 阿部洋丈，加藤和彦，王　維：セキュリティポリシーの動的切替機構を持つリファレンスモニタシステム，コンピュータシステム・シンポジウム論文集，pp.61–68 (2002).
24) 大山恵弘，神田勝規，加藤和彦：安全なソフトウェア実行システム SoftwarePot の設計と実装，コンピュータソフトウェア，Vol.19, No.6, pp.2–12 (2002).
25) 品川高廣，河野健二，高橋雅彦，益田隆司：拡張可能コンポーネントのためのカーネルによる細粒度軽量保護ドメインの実現，情報処理学会論文誌，Vol.40, No.6, pp.2596–2606 (1999).

付　　録

A.1 実験に使用した訓練コマンド列

A.1.1 Apache httpd

(1) `apachectl start` を実行．サーバには Apache インストール時のドキュメント群を置く．
(2) `wget -r http://localhost:8080/` を実行
(3) `wget -r --wait=1 --timeout=3 --referer= http://www.osss.is.tsukuba.ac.jp/ http://localhost:8080/` を実行
(4) `apachectl stop` でサーバを停止．

A.1.2 ProFTPD

(1) `proftpd -n -d 5 -c PFTEST.conf` を実行．`PFTEST.conf` は ProFTPD 付属のテスト用設定ファイル．
(2) サーバと同じ計算機で `ftp -n -d` を実行．
(3) 起動された ftp プログラム内で以下のコマンドを実行．
  (a) `open localhost 2021`
  (b) `user proftpd`（後にパスワードを入力）
  (c) `ls`
  (d) `pwd`
  (e) `binary`
  (f) `cd PFTEST`
  (g) `ls`
  (h) `mget PFTEST.*`（応答は全部 y）
  (i) `get not_exist`
  (j) `put pot.tar.gz`
  (k) `pwd`
  (l) `cd ..` を 4 回実行
  (m) `ls`
  (n) `bye`
(4) シグナルを送ってサーバを停止．

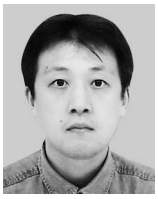
A.1.3 OpenSSH sshd

(1) `/etc/rc.d/init.d/sshd start` でサーバを開始．
(2) `ssh yosh@localhost` を実行．パスワードは 1 回目に正しいものを入力．ログインしたらすぐに exit を実行．
(3) `ssh yosh@localhost` を実行．1 回目のパスワード入力では誤ったパスワードを入力し，2 回目に正しいパスワードを入力．ログインしたらすぐに exit を実行．
(4) `scp yosh@localhost:/not_exist .` を実行（パスワードは 1 回目に正しいものを入力）．
(5) `ssh yosh@localhost` を実行．1 回目のパスワード入力を Ctrl-C で中断する．
(6) `scp yosh@localhost:/etc/hosts .` を実行（パスワードは 1 回目に正しいものを入力）．
(7) (2) を再度実行．
(8) `ssh yosh@localhost` を実行．3 回連続で誤ったパスワードを入力．
(9) `/etc/rc.d/init.d/sshd stop` でサーバを停止．

A.1.4 Emacs

(1) `emacs -q` で Emacs ウィンドウを出す．ウィンドウ内で以下の操作を実行．
(2) C-x C-f を入力．
(3) `/etc` を入力．Tab キーを押して補完．
(4) `/etc/sendmail.cf` をバッファに読み込む．
(5) 何も編集せず，C-x k で `sendmail.cf` をバッファから消す．
(6) C-x C-f で，新規ファイル `/tmp/foo` をバッファに読み込む．Tab キーによるタブ補完は使わない．
(7) `Hello world!` と 1 行書き込み，`/tmp/foo` を保存する．
(8) Emacs のタイトルバーの最小化ボタンを押す．
(9) デスクトップ画面下のツールバーのボタンを押して Emacs を元のサイズに戻す．
(10) C-x C-c で Emacs を終了．

(平成 14 年 12 月 21 日受付)
(平成 15 年　4 月 13 日採録)



大山 恵弘（正会員）

1973 年生．2001 年東京大学より博士（理学）取得．2001 年から 2003 年まで科学技術振興事業団研究員として筑波大学に勤務．2003 年より東京大学情報理工学系研究科助手．情報処理学会平成 13 年度論文賞受賞．興味はセキュリティ，システムソフトウェア，プログラミング言語，並列分散処理．

王 維

1974 年生．1997 年中国北京工業大学計算機応用学科卒業．同年同大学ソフトウェア研究所入社．2001 年筑波大学理工学部理工学科卒業．

加藤 和彦（正会員）

1962 年生．1989 年東京大学理学部情報科学科助手，1993 年筑波大学電子・情報工学系講師，1996 年同助教授，現在に至る．1992 年博士（理学）．1997 年より科学技術振興事業団さきがけ研究 21「情報と知」領域研究員，2000 年より同「協調と制御」領域研究員．2002 年より国立情報学研究所客員助教授．オペレーティングシステム，分散システム，プログラミングシステム，データベースシステムの研究に従事．