情報処理学会論文誌　コンピューティングシステム　Vol. 2　No. 1　12–22 (Mar. 2009)

# コールスタック情報を利用したモデル分割に基づく異常検知システム

神山貴幸[†1]　大山恵弘[†1]

アプリケーションの脆弱性を突く攻撃を防止するために，正常な実行におけるシステムコールのパターンを用いた異常検知システムが数多く提案されてきた．しかし，それらの多くはプログラム全体で 1 つの正常動作のモデルを作成しているため，プログラムの異なるフェーズにおけるシステムコール呼び出しの情報が融合され，検知精度が下がるという問題がある．そこで本研究では，モデル分割に基づく異常検知システムを提案する．このシステムは実行時のコールスタックの情報を用いてプログラムの実行をフェーズに自動的に分割し，フェーズごとに正常動作のモデルを作成する．また，実行状態に応じて動的に正常動作のモデルを変更する．提案システムは Linux 上に実装し，検知精度の 1 つの尺度である branching factor とオーバヘッドを測定し，既存システムとの比較を行った．

# Anomaly Detection System Based on Model Partitioning Using Call Stack Information

Takayuki Kamiyama[†1] and Yoshihiro Oyama[†1]

Many anomaly detection systems have been proposed to prevent attackers from exploiting vulnerabilities in applications. An important class of the systems is ones that utilize system call patterns of normal application behavior. Unfortunately, many of them have limitations in detection accuracy. One reason is that they merge system call information in different execution phases and create only one normal behavior model for an entire program. In this work, we propose an anomaly detection system based on a model partitioning technique. The system automatically partitions the execution of a given program into multiple phases using call stack information, and creates a normal behavior model for each phase. It dynamically changes the normal behavior model according to its execution state. We implemented the proposed system on Linux and compared it with an existing system in terms of branching factors and overheads.

## 1. はじめに

悪意ある者によるアプリケーションの脆弱性を突いた攻撃が後を絶たない．アプリケーションの脆弱性は，プログラムのバグに起因しており，実行前にそれらをすべて発見し，取り除くことは非常に困難である．そこで，アプリケーションの異常を早期に発見し対策を講じるための手段として，異常検知システムが提案されている．

異常検知システムは，あらかじめアプリケーションの正常動作を表現する**モデル**を作成し，アプリケーションがそのモデルに合った動作をするかどうか実行を監視する．このとき，モデルに合わない動作が観測されると，それを異常と見なし，ユーザに警告を出したり，実行を終了させたりするなどの処理を行う．異常検知システムは正常動作を表現するモデルを利用するため，未知の攻撃も検知できるという特徴を持っている．

モデルを作成するための正常動作の情報には，アプリケーションが発行するシステムコールの履歴がよく用いられる[2),3),6),8),11)]．正常動作時に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を一般化することで，プログラムの制御フローの規則を生成する．この手法は，バッファオーバフロー攻撃のようにプログラムの制御を奪い，任意のコードを実行可能にする攻撃に対して有効である．仮に，攻撃コードによって悪意あるシステムコールが発行されたとしても，システムコールのパターンが規則に合わなければ異常として検知される．しかし，そのような検知手法を回避するために，正常なシステムコールのパターンを偽装する攻撃[4),7),11),12)] も存在する．このような攻撃を防ぐには，攻撃の偽装を困難にする高精度なモデルを作成することが効果的である．

しかし，既存の異常検知システムの多くは，プログラム全体の動作を 1 つのモデルで表現しているため，実行状態の異なるシステムコール呼び出しの情報が融合されてしまうという問題がある．その結果，検知精度が下がり，偽装攻撃によって検知を回避されやすいモデルとなっている．

本研究では，正常動作のモデルを分割し，プログラムの実行の状態に応じたモデルを生成する異常検知システム PhaIDS を提案する．PhaIDS では，Sekar らの手法[8)] を改良し，実行時のオーバヘッドを考慮しながら，より検知精度の高いモデルを生成する．彼らの手法は，システムコールのパターンを有限オートマトンのモデルで表現し，プログラム全体で 1

†1 電気通信大学大学院電気通信学研究科情報工学専攻
Department of Computer Science, Graduate School of Electro-Communications, The University of Electro-Communications

つの巨大なオートマトンを作成する．PhaIDS では，プログラムの実行中の関数の情報が格納されているコールスタックの情報を利用して，オートマトンを複数に分割し，このオートマトンの集合をモデルとして用いる．そして，プログラムの実行の状態に応じて，動的にオートマトンを切り替えることで，それぞれの状態に合った粒度の細かい異常検知を実現する．また，アプリケーションごとに適した状態分けを可能にし，実行時のスタック検査を効率的に行うことで，検知によるオーバヘッドを軽減させている．なお，PhaIDS では，スタック検査が正確に行えることを前提としているため，監視の対象は C 言語の呼び出し規約に従ったプログラムに限定している．本研究では，PhaIDS を Linux 上に実装し，システムの有効性を検証するために，生成されるモデルのサイズ，検知精度，オーバヘッドについて実験を行った．

本論文は，以下のように構成される．2 章では，既存の異常検知システムとその問題点について述べる．3 章で PhaIDS の設計について述べ，4 章でその実装について述べる．5 章では PhaIDS への攻撃に対する耐性について議論し，6 章で実験結果を示す．7 章で関連研究について述べ，8 章で本論文をまとめる．

## 2. 既存の異常検知システム

アプリケーションのシステムコールのパターンをモデル化する異常検知システムの研究は，これまでに数多くなされてきた．ここではまず，その中の代表的な研究の 1 つである Sekar らの研究[8] について説明する．

Sekar らの研究では，アプリケーションが発行するシステムコールの順序関係を**図 1** のような有限オートマトン（FSA）によってモデル化している．また，アプリケーションの正常動作の情報には，発行したシステムコールの種類に加え，そのプログラムカウンタの値も使用している．このオートマトンの各ノードは呼び出したシステムコールの種類とプログラムカウンタの情報を保持しており，枝は次に呼び出すシステムコールのノードに対して伸びている．したがって，このオートマトンにおけるノード間の状態遷移は，アプリケーションによるシステムコールの発行によって行われる．状態遷移の際にはシステムコールの種類とプログラムカウンタのペアをチェックして遷移先ノードを探す．このようにしてオートマトンのノードをたどりながら実行を監視し，次の遷移先ノードが見つからない場合を異常として検知する．プログラムカウンタの情報を加えることで，条件分岐などによる異なる場所での同じ種類のシステムコール呼び出しを区別している．

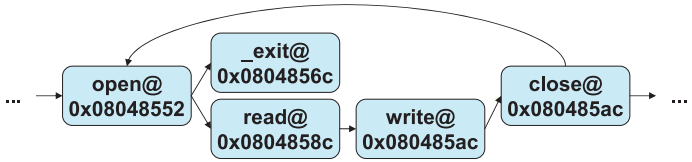


**図 1**　Sekar らの手法によって作成されるオートマトンの例
Fig. 1　Example of automaton created by Sekar, et al.'s method.

上記の方法で作成されたモデルでは，アプリケーションの動作の情報を正確に表現できない場合が存在する．具体的には，アプリケーション内の異なるプログラム部分によって行われたシステムコールの情報が，1 つのノードに融合されてしまう．これを**図 2** に示すプロ

```
main() {
  init();
  work();
}

init() {
  write_message();
  read_passwd();
}

work() {
  write_message();
  read_datafile();
}
```

```
write_message() {
  write(...);
}

read_passwd() {
  open("/etc/passwd");
  ...
}

read_datafile() {
  open("foo");
  ...
}
```

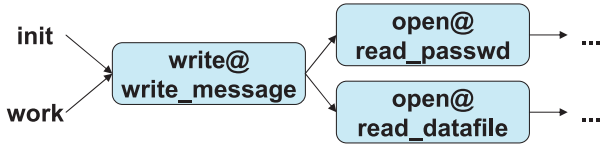


**図 2**　Sekar らの手法で動作情報を正確に表現できないプログラムの例
Fig. 2　Example of a program the behavior of which cannot be accurately expressed by Sekar, et al.'s method.

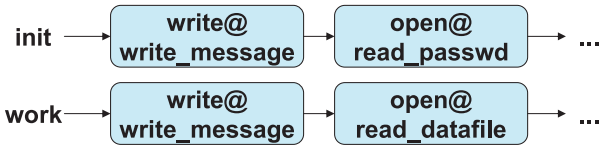


**図 3**　理想的なオートマトン
Fig. 3　Ideal automaton.

グラムを例にとって説明する．このプログラムは，プログラムの初期化を行う関数 `init` と主要な処理を行う関数 `work` によって構成されており，一般的なプログラムの処理の流れを単純化して表現している．また，このプログラムでは，初期化処理においてのみパスワードファイルを読むが，以降の処理では読まない．特にこの例では，いずれの関数も同じ関数 `write_message` を呼び出している場合について扱う．いずれも，まず関数 `write_message` の中で `write` を呼び出す．次に，関数 `init` は関数 `read_passwd` の中で，関数 `work` なら関数 `read_datafile` の中で，それぞれの `open` システムコールを呼び出す．これを Sekar らのモデルで表現すると図 2 の下の図のオートマトンのようになる．図 2 を見て分かるように，関数 `write_message` 内の `write` の呼び出しにおいて，関数 `init` からの呼び出しの情報と関数 `work` からの呼び出しの情報が融合されてしまっている．そのため，仮に関数 `work` から関数 `write_message` 内の `write` を呼び出した後，関数 `read_datafile` ではなく関数 `read_passwd` の `open` の呼び出しが試みられても，このモデルでは正常として判断されてしまう．本来ならば，このような呼び出しは異常として検知されるべきである．そのためには，**図 3** のように，それぞれの関数からの呼び出しの情報に応じて状態を分けなければならない．しかし，Sekar らの手法を含め多くの既存の異常検知システム[3),6),8),11)] は，複数のシステムコール呼び出しの情報を 1 つの状態で表現しているため，このような呼び出しを区別することができない．この問題は，プログラムの各状態における許容動作の範囲を広げ，攻撃を見逃す機会を増やすことにつながる．

このような問題に対して，Feng らの研究[2)] では，コールスタックの情報を利用したモデルを生成している．彼らのモデルでは，2 つの連続したシステムコール呼び出しの間で呼ばれる関数の履歴を得るために，それら 2 つの呼び出しのときのスタックの差分を取り出し，**図 4** のような VtPath と呼ばれる仮想的な関数の遷移のパスを生成する．そして，VtPath の集合を正常動作のモデルとして用いる．このモデルを用いると，システムコール呼び出しとそのときのコールスタックの状態に応じた細かな状態分けが可能である．しかしその反面，仮想的な関数の遷移パスを生成するために，コールスタックをシステムコール呼び出しのたびに遡らなければならないため，オーバヘッドが大きいという問題点がある．

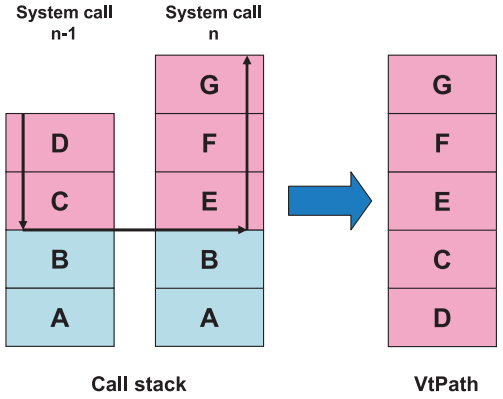


**図 4**　VtPath の例
Fig. 4　Example of VtPath.

## 3. 異常検知システム PhaIDS

### 3.1 概　　要

PhaIDS は，コールスタックの情報を用いてアプリケーションの実行を**フェーズ**という単位に分割し，そのフェーズごとに正常動作のモデルを自動生成する．そして，実行時にコールスタックの状態を参照して現在の実行フェーズを調べ，そのフェーズに対応するモデルに動的に切り替えながらアプリケーションの監視を行う．正常動作のモデルには Sekar らのオートマトンのモデルを使用する．ただし，この 1 つの巨大なモデルを複数のモデルに分割し，1 つ 1 つのモデルを小さくすることで，アプリケーションの実行の各状態における許容動作の範囲を狭める．これにより，2 章で述べた複数のシステムコール呼び出しの情報が 1 つのノードに融合されてしまうという問題を解決する．また，フェーズの分割によってモデルの状態を適切な数に分割し，過剰な状態分けを減らすことで，コールスタック検査によるオーバヘッドを軽減させる．なお，PhaIDS の利用において異常が検知された場合には，監視対象のアプリケーションの管理者に異常の旨の警告メッセージを送り，早期な対策を促すことを想定している．

### 3.2　正常動作情報の収集

まず，正常動作のモデルを生成するために，あらかじめ監視対象のアプリケーションを実行させて，その動作情報を記録する．この実行中には攻撃は行われないと仮定する．アプリケーションがシステムコールを呼び出すたびに実行を停止させ，システムコールの種類とライブラリ関数に入る直前のプログラムカウンタを動作情報として記録する．プログラムカウンタの値はそのときのコールスタックから取得する．このとき，コールスタックには関数フレームとして，現在呼び出し中の関数の情報が各関数が呼ばれた順に積まれているので，各フレーム内のプログラムカウンタの値がアプリケーションコードの領域内かどうかをチェックしながら，スタックを遡って，ライブラリ関数を呼び出しているプログラムカウンタを取得する．Sekar らの手法では，以上の 2 つの情報を記録するだけであるが，本研究ではさらに現在のシステムコール呼び出しに至るまでの関数呼び出しの情報，すなわち，各関数フレームに保存されたリターンアドレスの列を記録する．

### 3.3　フェーズ分割とモデル生成

前節のようにしてアプリケーションの正常動作情報を収集したら，この情報を基にアプリケーションの実行をフェーズに分割する．このフェーズ分割は，アプリケーションが行う処理内容に応じて分割されることが望ましい．たとえば，図 2 の例では，初期化処理（関数 `init` による呼び出し）においてのみ `/etc/passwd` のような重要なファイルへのアクセスが行われ，他の部分の処理（関数 `work` による呼び出し）ではそのような重要な処理を含まない．このような場合，アプリケーションの処理の変化に沿ってフェーズを分割できれば，重要な処理を行うフェーズを適切に限定することができる．

そこで PhaIDS では，**図 5** のような複数の小さな Sekar らのオートマトンから構成される 1 つの巨大なオートマトンのモデルを生成する．アプリケーション全体としては 1 つの巨大なオートマトンだが，各実行フェーズにおいては，対応する小さなオートマトンに定義された動作しか許可されない仕組みとなっている．なお，フェーズはアプリケーションの実行状態を表現しており，図 5 の Phase4 から Phase2 への遷移に示されているように，1 度抜けたフェーズに再び入ることもある．

PhaIDS では，実際のアプリケーションの処理の変化に沿ったフェーズ分割に近づけるために，アプリケーションが現在行っている処理をコールスタックに積まれた関数フレームから推定し，関数フレームの変化によってフェーズを分割している．つまり，あるシステムコール呼び出しのときにコールスタックに積まれていた関数フレーム群が，次のシステムコール呼び出しのときで大きく変化していたら，アプリケーションの処理すなわちフェーズ

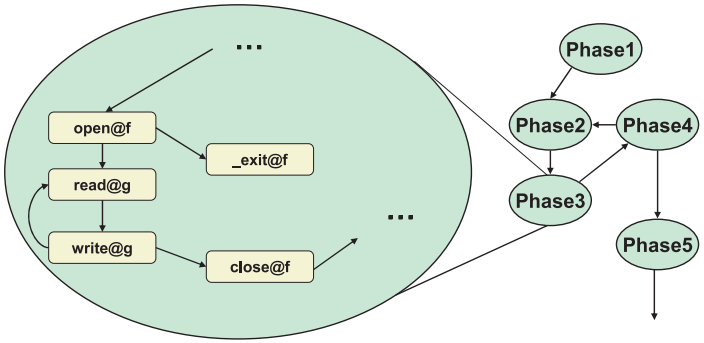


**図 5**　PhaIDS によって生成されるモデルの例
Fig. 5　Example of a model created by PhaIDS.

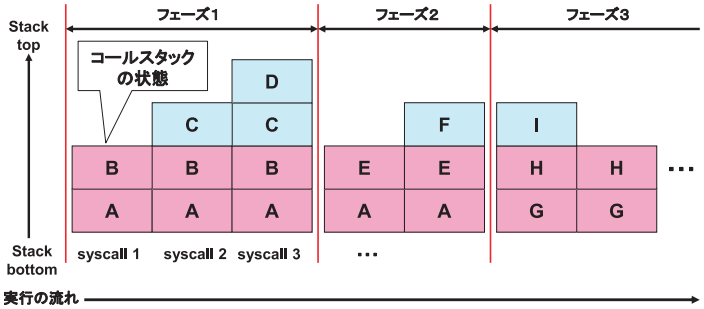


**図 6**　比較フレーム数を 2 に設定したときのフェーズ分割の例
Fig. 6　Example of phase partitioning when the number of compared frames is set to two.

が変化したと考え，そこでフェーズを分割する．関数フレームの変化の度合いは，コールスタックのボトム側のフレームから判断する．具体的には，比較する関数フレームの数（以降，**比較フレーム数**）を設定し，コールスタックのボトムからその個数のフレームを比較して，それらすべてが一致するかどうかでフェーズを分割している．**図 6** は，ボトムからの比較フレーム数を 2 に設定してフェーズ分割した例である．ボトムの関数フレーム群が A-B，A-E，G-H というグループに分かれており，関数フレーム群が変化した時点でフェーズ分割されている．また，**図 7** のようにループの実行などの理由から，以前観測されたボトムの関数フレーム群（A-E）が別のフェーズを挟んで再び現れた際（フェーズ 4 の後には，現

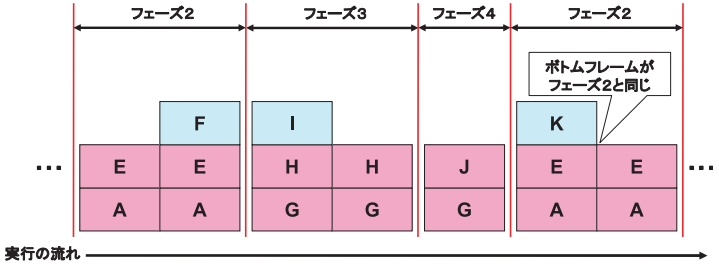


**図 7**　過去に訪れたフェーズに再び入る実行の例
Fig. 7　Example of execution that enters again a previously visited phase.

在の状態を前回現れたときと同一のフェーズ（フェーズ 2）に属させる．このとき，ボトムの関数フレーム群が同じであれば，それより上で呼び出す関数が以前（関数 F）と異なる関数（関数 K）を呼び出していてもかまわない．

プログラム中に再帰関数が存在する場合，実行時に同じ関数が繰り返し呼ばれ，コールスタックに同じ関数のフレームが重複して積まれることがある．この重複した関数フレームの数は，再帰呼び出しの回数によって変化するため，比較フレーム数によっては，実行のたびに異なるフェーズが観測されてしまうことになる．これは，実行の監視時において誤検知を引き起こす原因となる．そこで PhaIDS では，重複する関数フレームが連続してコールスタックに積まれている場合，それらを 1 つのフレームと見なすようにしている．

現在，比較フレーム数の設定は，PhaIDS のユーザが適切なフレーム数を選択する形をとっている．なお，PhaIDS では各フレーム数ごとの分割で生成されるモデルの検知能力を数値化し，フレーム数の選択を手助けする情報として提供している．モデルの検知能力の数値化には，モデルの検知精度の 1 つの尺度である branching factor [11] を採用している．Branching factor とは，モデルの各ノードにおいて異常と見なされずに発行できるシステムコールの数（遷移の種類数）の平均値である．この値が小さいほど，攻撃者にとって検知を回避することが困難になる．

このようにして，アプリケーションの実行をフェーズに分割したら，Sekar らの手法にならって各フェーズごとにオートマトンのモデルを生成していく．各フェーズのモデルを生成したら，フェーズ間の状態遷移のために，各フェーズで生成されたオートマトンの終端ノード（フェーズの出口）から次のフェーズのオートマトンの初期ノード（フェーズの入口）に枝を加えていく．

### 3.4　実行の監視

PhaIDS は，前節の方法で生成されたモデルを基に，アプリケーションの実行を監視する．監視の場合も，正常動作情報の収集と同じようにアプリケーションがシステムコールを呼び出すたびに実行を停止させる．そして，呼び出したシステムコールの種類とそのプログラムカウンタの値をチェックする．もし，現在いるノードが現在のフェーズの出口ならば，フェーズ分割のときに設定された比較フレーム数だけコールスタックのボトムの関数フレームを調べる．一方，現在いるノードがフェーズの出口でなければ，コールスタックのボトムは調べない．スタック検査の処理はオーバヘッドを増加させるので，フェーズ間の状態遷移の可能性があるノードだけに検査を絞ることによって，極力スタック検査の回数を減らすようにしている．以上のようにして，現在いるノードから，対応する遷移先があるかどうかチェックし，遷移先が見つかれば状態を遷移させる．見つからない場合は，異常と見なしてまず警告を出し，コールスタックを検査する．

スタック検査の結果，フェーズが遷移していなければ，現在のフェーズのオートマトンからシステムコール呼び出しに対応するノードを探す．ノードが見つかれば，それに状態を遷移させる．ノードが見つからない場合は，そのフェーズ内で未知ノードと呼ばれる特殊な状態に遷移させ，実行を続ける．ただし，この未知ノードにいる間は，システムコール呼び出しのたびに警告を出す．未知ノードから通常のノードへの復帰は以下のように行う．未知ノードにいる状態でシステムコールが呼び出されるたびに，その呼び出しに対応するノードがあるかどうか，オートマトンを検索する．該当するノードが見つかれば，未知ノードからそのノードに状態を遷移させる．

スタック検査の結果，フェーズが遷移しているなら，遷移先のフェーズに対応するオートマトンを探す．オートマトンが見つかれば，そのオートマトンから現在のシステムコール呼び出しに対応するノードを探し，見つかればそれに遷移させる．ノードが見つからなければ，そのフェーズ内の未知ノードに状態を遷移させる．もし，遷移先のフェーズに対応するオートマトンが見つからなければ，未知フェーズという特殊なフェーズの中の未知ノードに状態を遷移させる．この未知フェーズでは，システムコール呼び出しのたびに警告を出す．未知フェーズから通常のフェーズへの復帰は以下のように行う．未知フェーズにいる状態でシステムコールが呼び出されるたびに，スタックのボトムの関数フレーム群を検査する．もしその関数フレーム群がモデルに存在するフェーズのものであったら，そのフェーズに対応するオートマトン内のその呼び出しに対応するノードに状態を遷移させる．

## 4. 実　　装

### 4.1 監視方法

PhaIDS は Linux 上に実装されている．PhaIDS は監視の開始の際に，プロセスを生成する．親プロセスが異常検知を行い，子プロセスがアプリケーションを実行する．親プロセスは，監視対象のアプリケーションを `ptrace` システムコールを用いて監視している．アプリケーションがシステムコールを呼び出すたびに，`ptrace` システムコールによって実行が停止され，親プロセスに制御が移る．このとき，親プロセスは `ptrace` システムコールによって，子プロセスのレジスタの値を取得して，システムコールの種類を判別したり，子プロセスのアドレス空間内のデータを読み取ることで，スタックの情報を取得したりしている．

### 4.2 fork/exec への対応

監視対象のアプリケーションが，実行中に `fork` システムコールなどを呼び出して新たな子プロセスを生成した場合，監視側のプロセスも同様に新たなプロセスを生成し，新たなアプリケーションの子プロセスの監視に割り当てるようにしている．また，子プロセスの監視に用いるモデルは `fork` システムコールを呼び出したアプリケーション内のプログラムカウンタごとに専用のモデルを生成するようにしている．つまり，親プロセスとは別のオートマトンになっており，子プロセスどうしであっても生成場所ごとにオートマトンは異なる．また，アプリケーションが実行中に `exec` システムコールを呼び出した場合には，正常動作のモデルを `exec` システムコールの引数のパスに指定されたプログラムのものに切り替える．

### 4.3 モデルのデータ構造

PhaIDS では，アプリケーションの正常動作のモデルの情報を以下の 3 種類のテーブルによって保持しており，それらは**図 8** のような階層構造となっている．

- プロセスモデルテーブル
  アプリケーションが生成するプロセスごとのモデルの情報を保持するためのテーブル．各プロセスのモデルの ID とフェーズテーブルとの対応付けを行う．モデルの ID には，そのプロセスを生成する `fork` システムコールのプログラムカウンタの値を使用している．
- フェーズテーブル
  モデルの各フェーズのボトムフレームとそれに対応するオートマトンの情報を保持するためのテーブル．各フェーズのボトムフレームと FSA テーブルとの対応付けを行う．

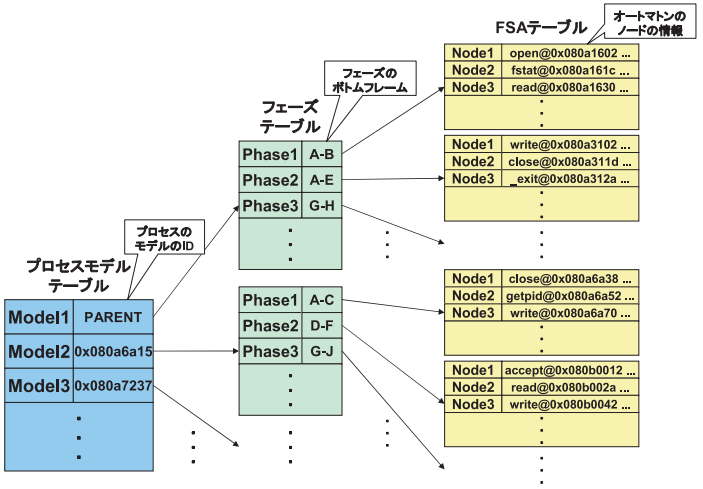


**図 8**　PhaIDS におけるモデルのデータ構造
Fig. 8　Data structure for models in PhaIDS.

- FSA テーブル
  オートマトンの各ノードの情報を保持するためのテーブル．各ノードごとにシステムコール番号，プログラムカウンタ，フェーズの出口ノードか否かの情報，遷移先候補リストを格納している．遷移先候補リストは，現在のノードから遷移先候補の各ノードへのポインタを格納したリストである．

実行時には，各テーブルに対して，現在の状態に対応する項目（モデル，フェーズ，ノード）へのポインタを保持しながら監視を行っている．オートマトンでの状態遷移の際には，ポインタによって FSA テーブルの現在のノードの項目にアクセスし，その遷移先リストから呼び出しに対応する遷移先ノードを探し，見つかれば現在のノードへのポインタを遷移先ノードへのポインタで上書きする．フェーズが遷移する際には，フェーズテーブルへのポインタも新しいフェーズのものに上書きする．また，`fork` システムコールによって子プロセスが生成された場合には，そのプログラムカウンタの値を使ってプロセスモデルテーブルからその子プロセスに対応するモデルを探す．そして，フェーズへのポインタを該当するモデルの持つフェーズテーブルの初期フェーズにセットし，現在のノードへのポインタもその初期フェーズに対応する FSA テーブルの初期ノードにセットすることで，子プロセスの監視

を開始する．

## 5. 議　　論

システムコールだけを検査する異常検知システム[3),11)]でも，単純に望みのシステムコールを呼び出すだけの攻撃であれば高確率で検知が可能である．しかし，正常なシステムコールパターンにマッチするように攻撃コードを偽装することによって検知をすり抜けた例[4),7),11),12)]も報告されている．この偽装攻撃は，正常なシステムコールパターンにマッチするようにシステムコールを呼び出すようにすればよいので，あらかじめそのパターンを知っていれば比較的容易に成功させることができる．

Sekar らの手法では，システムコールとそのプログラムカウンタの値を用いるため，このような攻撃も検知可能である．しかし，システムコールのプログラムカウンタの値まで偽装するような攻撃を検知することはできない．

PhaIDS では，フェーズが切り替わるたびにコールスタックのボトムの検査を行っているので，正常なシステムコールパターンとプログラムカウンタを偽装しただけでは検知を回避することは困難になっている．仮に，システムコールのプログラムカウンタの値までを偽装したとしても，現在のフェーズのモデルに合うシステムコールしか呼び出すことができないので，攻撃における許容動作は限定される．

コールスタックの情報を偽装することにより，PhaIDS の検知を回避する攻撃は可能である．ただし，この攻撃では，モデルに合うシステムコール呼び出しとコールスタックのリターンアドレスおよびフレームポインタを偽装する必要があり，前述の単純な偽装攻撃よりも複雑な処理を要する．さらに，PhaIDS において攻撃者が望みのシステムコールを呼ぶには，そのシステムコールを呼び出すフェーズまでフェーズを遷移させる必要があり，フェーズ遷移のたびに繰り返しスタックを偽装し続けなければならず，攻撃の難易度は格段に上がる．したがって，PhaIDS は攻撃の手間を増やし，敷居を高くすることによって，攻撃者の意欲を削ぎやすくしており，攻撃の検知だけでなく攻撃抑止の効果も期待される．

## 6. 実　　験

PhaIDS の有効性を検証するために，PhaIDS によって生成されるモデルのサイズ，検知精度，オーバヘッドを測定する実験を行った．実験環境は，Intel Celeron D（3.20 GHz），768 MB RAM，Ubuntu 7.10（kernel 2.6.22）である．アプリケーションは，POP サーバの Qpopper 4.0.13 と Web サーバの Apache 2.2.8 を使用した．

### 6.1 モデルのサイズ

各アプリケーションに対して以下のような操作を行い，モデルを生成した．Qpopper に対しては，1 回の接続で USER，PASS，QUIT などを含めた 3〜7 個のコマンド列をランダムに生成して送信し，それを 100 回繰り返した．Apache に対しては，我々の研究室の Web ページをミラーリングしてドキュメントディレクトリにファイルを置き，その Web サーバのログから抽出したリクエスト情報を用いて，`wget` コマンドによりリクエストを送信した．なお，サーバのログから抽出した 195,267 リクエストのうち，6 割をモデルの生成に使用し，残り 4 割を誤検知の数を測定する実験に用いた．以上の操作によるアプリケーションの動作を記録して，比較フレーム数を変えながらそれぞれモデルを生成し，子プロセスのモデルも含めたすべてのプロセスのモデルのオートマトンの数，ノードの数，枝の数，サイズをそれぞれ計算した．

なお，本実験で Qpopper と Apache に対して行った操作は，実際の利用シチュエーションで行われる操作のパターンの多くを含んでいると我々は考えている．しかし，実際の利用では，今回行っていない操作のパターンをユーザが実行することは十分に考えられる．よって，本実験による評価は，ユーザが行う可能性がある操作のパターンすべてを網羅していないという限界がある．実際の利用シチュエーションにおいて操作の履歴を収集し，それに基づいてさらに詳細な評価を行うことは今後の課題である．

Qpopper，Apache における各結果をそれぞれ**表 1**，**表 2** に示す．それぞれの結果において，比較フレーム数が 1 のときは，プロセスごとのモデルはフェーズ分割されていない状態なので，Sekar らのモデルと等価のモデルとなっている．どちらも，比較フレーム数が増えるにつれ，Sekar らのモデルよりノードの数，枝の数がともに増加していることが確認できる．これは，Sekar らのモデルの中にシステムコール呼び出しの情報が融合されたノードが複数存在し，モデルの分割によってそれらの状態がうまく分けられたことを示している．その結果，ノードや枝の数に応じてモデルのサイズも増加している．

モデルが最も多く分割されるのは，比較フレーム数の値がアプリケーションの実行中に一番多く積まれたときのフレーム数となったときである．なお，このときのモデルは，Feng らの VtPath と同程度に状態分けされたモデルとなっている．Qpopper の場合は最大 8 フレーム，Apache の場合は最大 21 フレームであった．しかし，Qpopper は比較フレーム数が 5 以上，Apache は比較フレーム数が 13 以上になったときから，ノードの数，枝の数に変化が見られなくなった．これは，さらに比較フレーム数を増やしてフェーズを分割しても，状態を分けられるノードが存在しなくなったことを表している．したがって，比較フ

**表 1**　Qpopper のモデルのサイズ
Table 1　Sizes of models for Qpopper.

| 比較フレーム数 | 1 | 4 | 5 | 8 |
|---|---|---|---|---|
| オートマトンの数 | 4 | 20 | 36 | 41 |
| ノードの数 | 196 | 219 | 224 | 224 |
| 枝の数 | 259 | 288 | 297 | 297 |
| サイズ [KB] | 3.89 | 4.79 | 5.42 | 6.09 |

**表 2**　Apache のモデルのサイズ
Table 2　Sizes of models for Apache.

| 比較フレーム数 | 1 | 3 | 5 | 9 | 13 | 21 |
|---|---|---|---|---|---|---|
| オートマトンの数 | 3 | 35 | 69 | 104 | 132 | 136 |
| ノードの数 | 263 | 349 | 354 | 375 | 387 | 387 |
| 枝の数 | 483 | 668 | 672 | 693 | 706 | 706 |
| サイズ [KB] | 6.19 | 9.07 | 10.4 | 13.3 | 16.9 | 21.4 |

レーム数を増やしても過剰にフェーズ分割されるだけで，各フェーズのモデルで保持される関数フレーム群の情報が増え，モデルのサイズだけが増える結果となった．

### 6.2 検知精度

前節で生成した各モデルの検知精度を評価するために，検知精度の 1 つの尺度である branching factor [11] と異常検知での誤検知（false positive）の数を測定した．

Branching factor は，モデルの各ノードにおいて異常と見なされずに発行できるシステムコールの数（遷移の種類数）の平均値であるため，計算方法は，表 1，表 2 を基にして，オートマトンの枝の数をノードの数で割ればよい．ただし，完全に分割されていないモデルでは，ノードから出る枝の数とそのノードからの実際の遷移の種類数は異なるため，branching factor の計算にはフェーズ分割数が最大のときのモデルの枝の数を用いる．Qpopper，Apache のそれぞれの結果を**表 3**，**表 4** に示す．表 3，表 4 から，Qpopper では比較フレーム数 4，Apache では比較フレーム数 3 のときにおいて，branching factor の値が大きく減少していることが確認できる．したがって，Sekar らのモデルに比べ，検知を回避されにくいモデルとなっていることが分かる．また，Qpopper においては比較フレーム数 5 と 8，Apache においては比較フレーム数 13 と 21 に設定したときで，同程度の検知能力を持ったモデルとなっていることが分かる．つまり，比較フレーム数最大まで分割しなくても，Feng らの VtPath と同程度の検知能力が得られるということである．

**表 3**　Qpopper における branching factor と誤検知の数
Table 3　Branching factors and false positives in the experiment of Qpopper.

| 比較フレーム数 | 1 | 4 | 5 | 8 |
|---|---|---|---|---|
| Branching factor | 1.52 | 1.36 | 1.33 | 1.33 |
| 誤検知の数 | 3 | 5 | 10 | 10 |

**表 4**　Apache における branching factor と誤検知の数
Table 4　Branching factors and false positives in the experiment of Apache.

| 比較フレーム数 | 1 | 3 | 5 | 9 | 13 | 21 |
|---|---|---|---|---|---|---|
| Branching factor | 2.68 | 2.02 | 1.99 | 1.88 | 1.82 | 1.82 |
| 誤検知の数 | 3 | 9 | 9 | 9 | 9 | 9 |

異常検知における誤検知の数を測定するために，6.1 節に述べた方法で生成したモデルを用いて各アプリケーションを監視し，それぞれについて以下の操作を行った．まず，Qpopper に対しては，モデルを生成するときと同じようにランダムに 100 接続分のコマンド列を生成し，これらをすべて送信したときに出た警告の数を測定した．なお，測定中に Qpopper が発行したシステムコールの回数は 298,288 であった．この結果を表 3 にあわせて示す．表 3 から，比較フレーム数が増えるにつれ，誤検知が増加していることが分かる．これは，branching factor の値からもいえるように，モデル分割によってシステムコール呼び出しの状態が分けられ，各状態から伸びる枝の数が減少し，検知の条件が厳しくなったためである．Apache に対しては，サーバのログから抽出したリクエストのうち，モデルの生成に使用しなかった残り 4 割のリクエストを使用して，同様に `wget` コマンドでリクエストを送信し，このときの警告の数を測定した．なお，測定中に Apache が発行したシステムコールの回数は 3,352,239 であった．この結果を表 4 にあわせて示す．表 4 より，Apache では比較フレーム数が 1 から 3 の間で誤検知の数がわずかに増加したが，それ以降は比較フレーム数が増えても誤検知の数に変化は見られなかった．この誤検知の原因は，モデル生成のための実行で記録されなかったシステムコール呼び出しが行われたことや，フェーズ分割に関係ない状態遷移が新たに観測されたことによるものである．

一般的に異常の検知能力と誤検知の数はトレードオフの関係にある．PhaIDS においても，表 3，表 4 の結果に表れているように，トレードオフの性質を持っている．前述のように，PhaIDS では，比較フレーム数最大までフェーズを分割しなくても，最大のときと同程度の検知能力が得られる場合がある．しかし，検知・誤検知のトレードオフの関係から誤検知率も同程度となってしまう．そのため，ユーザが検知能力と誤検知の数の関係を考慮し

**表 5**　PhaIDS により監視された Apache のリクエスト処理時間
Table 5　Amount of time take for processing a request by an Apache server monitored by PhaIDS.

| PhaIDS | 1 リクエストあたりの処理時間 [ms] | オーバヘッド [%] |
|---|---|---|
| 未導入 | 1.162 | - |
| 導入：監視なし | 1.482 | 27.5 |
| 導入：比較フレーム数 1 | 1.625 | 39.8 |
| 導入：比較フレーム数 3 | 1.639 | 41.0 |
| 導入：比較フレーム数 5 | 1.671 | 43.8 |
| 導入：比較フレーム数 9 | 1.881 | 61.9 |
| 導入：比較フレーム数 13 | 1.925 | 65.6 |
| 導入：比較フレーム数 21 | 1.954 | 68.1 |
| VtPath | 1.964 | 69.0 |

て適切な比較フレーム数を設定することが重要となっている．一般に，比較フレーム数を大きくすると，検知能力は上がるが，誤検知の数が増える．比較フレーム数を小さくすると，逆の結果になる．適切な比較フレーム数を用いると，検知能力をほとんど低下させずに誤検知の数を減らせる場合がある．たとえば，表 3 における比較フレーム数 4 と 5 のときで，branching factor の値はほとんど変わらないが，比較フレーム数を 4 に設定すれば，誤検知の数は 5 に設定したときの半分に抑えられる．

さらに，Qpopper のモデルにおいて，処理内容に応じてフェーズがうまく分割できているかどうかを調査した．Qpopper の処理は初期化処理，ユーザ名取得処理，パスワード取得処理，メールを送信する処理などからなる．これらの各処理は独立性が高いため，これらを別々のフェーズとして扱うことにより，高精度の異常検知が可能になる．調査の結果，比較フレーム数を 4 に設定したときに，Qpopper が行う各処理の境界にきわめて近い部分でフェーズが分割されていることを確認できた．

### 6.3　オーバヘッド

PhaIDS による実行の監視によるオーバヘッドを検証するために，ApacheBench を用いて Apache のリクエスト処理時間を測定した．計測中は警告を出さないようにするために，6.1 節に述べた方法で生成したモデルに対して，さらに `ab -n 1000 -c 10`（同時接続数 10 で 1,000 リクエスト送信）を実行した動作情報によって拡張してから，同じコマンドで計測を行った．また，比較のために Feng らの VtPath を実装し，同様に計測を行った．

結果を**表 5** に示す．「導入：監視なし」は，システムコール呼び出し時にアプリケーションを停止させるが，何も検査せず，すぐにアプリケーションの実行を再開させたものである．この場合でもオーバヘッドは 27.5%あり，システムコール監視の導入によるオーバヘッドは大きいことが分かった．この原因は，PhaIDS が `ptrace` システムコールによって実装されており，監視対象のアプリケーションがシステムコールを呼ぶたびに，PhaIDS のプロセスに切り替わるため，コンテキストスイッチによるオーバヘッドが蓄積がすることであると考えられる．したがって，今後システムコール監視の実装方法を改善することでオーバヘッドが軽減されることが見込める．

比較フレームごとのオーバヘッドでは，比較フレーム数が 1 から 5 のときにかけてはあまり大きな増加はない．つまり，これらの比較フレーム数の間では，Sekar らのモデルと同程度のオーバヘッドで，より高精度な監視が行えるということである．比較フレーム数をさらに増やしていくと，比較フレーム数が 9 の時点でオーバヘッドが大きく増加している．これは，単純にフェーズ分割数と比較フレーム数の増加が主な原因であると考えられるが，スタック検査の回数の増加も原因の 1 つである．PhaIDS では，スタック検査をフェーズの出口に絞ることでその回数を減らしているが，フェーズ分割数が増えると，1 つのフェーズ内におけるオートマトンのノードの数が減少してしまうため，結果的にスタック検査が頻繁に行われてしまうことになる．また，Feng らの VtPath による監視のオーバヘッドは，それと同程度の状態分けとなるモデルを生成する比較フレーム数 21 のときと，ほぼ同じであった．PhaIDS では，比較フレーム数を 13 や 9 まで減らしても，VtPath と同程度の検知能力があるが，検知能力を維持したままオーバヘッドを削減できる．

## 7. 関 連 研 究

本章では，アプリケーションのシステムコールのパターンに基づく異常検知システムの関連研究とその他のセキュリティシステムの研究について述べる．

Hofmeyr らの研究[3] では，N-gram と呼ばれる手法により，定められた長さ $N$ のシステムコールの部分列の集合を，正常動作のモデルとして用いている．監視の際には，観測されたシステムコール列から．長さ $N$ のシステムコールの部分列を取り出し，その部分列のパターンがモデルに含まれているかどうかで異常を判定している．ただし，この方法はシステムコールの種類以外の情報を利用しないため，正常なシステムコールのパターンを偽装した攻撃によって，検知を回避されやすい．だが，提案手法と組み合わせることによって，検知を回避されにくいモデルに改良することが可能である．

Paid[5] は，アプリケーションのソースコードから，自動的にモデルを生成する異常検知システムである．Paid は，システムコール呼び出し時にコールスタックを検査したり，意味

のないシステムコールの呼び出しをアプリケーションコードに挿入したりすることにより，偽装攻撃の実現にかかる手間を増加させている．Paid のスタック検査は，各関数フレームに保存されたリターンアドレスが，テキスト領域の中のアドレスかどうかを検査するだけの単純なものである．よって，攻撃者はテキスト領域のアドレス範囲を推測できれば，偽装されたスタックを容易に構築できる．一方，PhaIDS に対して偽装攻撃を成功させるには，攻撃者はその時点におけるアプリケーションの実行状態に応じたスタックを構築する必要があり，大きな手間がかかる．

異常検知システム以外に，アプリケーションの不正な動作からシステムを守るための仕組みとして，サンドボックスがある．これは，アプリケーションが行うファイルなどの資源への操作の許可・不許可をあらかじめポリシとして記述し，それに基づいて実行を監視することで，ポリシに合わない不正な動作を防ぐ．本研究におけるアプリケーションの実行の状態に応じたモデル切替えと同様に，サンドボックスにおいてもポリシを分割して動的に切り替える研究がなされている．SubDomain [1] では，`change_hat` という独自のシステムコールをアプリケーションに挿入することで，ポリシを動的に切り替えている．Shioya らの研究[9] では，本研究と同様に実行時のコールスタックの情報を利用して，現在呼ばれている関数群を把握し，その状態に応じたポリシに動的に切り替えている．これらのシステムでは，動的なポリシ切替えによってアプリケーションの状態に応じた動作を制限できるが，状態ごとに適切なポリシを記述しなければならず，ユーザの負担が大きい．品川らの研究[10] では，ポリシの記述を簡略化するために，サーバアプリケーションの実行の状態を，通信メッセージ受信前の初期化フェーズと受信後のプロトコル処理フェーズに分けている．メッセージ受信前はリモートから攻撃を受けることはないと仮定し，初期化フェーズでの複雑な処理のポリシの記述をなくすことで，記述量を減らし，ポリシ記述を簡略化している．PhaIDS では，アプリケーションの実行を自動的にフェーズに分割して各フェーズごとの正常動作のモデルを自動生成しているので，ユーザによる記述の手間はかからない．一般に，サンドボックスと比べて，PhaIDS では，ポリシ記述の手間がかからないという利点があるが，その反面，正常動作の情報を収集するのに手間がかかることや，判断を誤ることがあるなどの欠点もある．このトレードオフをよく理解し，状況に応じて使い分けることが必要である．

## 8. ま と め

本論文では，実行時のコールスタックの情報を用いて，アプリケーションの実行をフェーズに分割し，フェーズごとに正常動作のモデルを生成する異常検知システム PhaIDS を提案した．Qpopper と Apache の異常検知を行う実験を通じて，PhaIDS では既存のシステムに比べて，攻撃者が検知を回避しにくくなっていることを確認した．PhaIDS が Apache のリクエスト処理時間に与えるオーバヘッドは，フェーズ分割に利用する関数フレームの数が少ないときには，40%〜44%であった．このオーバヘッドは `ptrace` システムコールを用いてシステムコール監視を実現していることが主な原因だと考えられる．`ptrace` に代わる高速なシステムコール監視機構の導入によってこのオーバヘッドは削減可能である．

今後の課題としては，第 1 に，多くの種類のアプリケーションで評価実験を行うことがあげられる．6 章ではサーバアプリケーションを用いた評価結果を示したが，クライアントアプリケーションに対する PhaIDS の有効性についても今後評価していきたい．第 2 に，PhaIDS が検知できる攻撃と検知できない攻撃についてさらに考察することがあげられる．5 章で述べたように，PhaIDS に対して偽装攻撃を成功させることは困難だが，不可能ではない．成功させるために攻撃者はどの程度の作業コストをかける必要があるのかについて，今後詳細な調査が必要である．第 3 に，フェーズ分割のためのより良い方式の提案があげられる．現在はスタックのボトムにある固定数のフレームの情報をフェーズ分割に用いている．今後，検査するフレーム数を可変にしたり，リターンアドレス以外の情報を利用したりするなどの方式も検討していきたい．

## 参 考 文 献

1) Cowan, C., Beattie, S., Kroah-Hartman, G., Pu, C., Wagle, P. and Gligor, V.: SubDomain: Parsimonious Server Security, *Proc. 14th USENIX Systems Administration Conference* (*LISA 2000*), pp.355–368 (2000).
2) Feng, H.H., Kolesnikov, O.M., Fogla, P., Lee, W. and Gong, W.: Anomaly Detection Using Call Stack Information, *Proc. 2003 IEEE Symposium on Security and Privacy*, pp.62–75 (2003).
3) 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).
4) Kruegel, C., Kirda, E., Mutz, D., Robertson, W. and Vigna, G.: Automating Mimicry Attacks Using Static Binary Analysis, *Proc. 14th USENIX Security Symposium*, pp.161–176 (2005).
5) Lam, L.C. and cker Chiueh, T.: Automatic Extraction of Accurate Application-Specific Sandboxing Policy, *Proc. 7th International Symposium on Recent Advances in Intrusion Detection* (*RAID 2004*), pp.1–20 (2004).
6) Mutz, D., Valeur, F., Vigna, G. and Kruegel, C.: Anomalous System Call Detection, *ACM Trans. Information and System Security*, Vol.9, No.1, pp.61–93 (2006).

7) Parampalli, C., Sekar, R. and Johnson, R.: A Practical Mimicry Attack Against Powerful System-Call Monitors, *Proc. 2008 ACM Symposium on Information, Computer and Communications Security*, pp.156–167 (2008).
8) Sekar, R., Bendre, M., Dhurjati, D. and Bollineni, P.: Fast Automaton-Based Method for Detecting Anomalous Program Behaviors, *Proc. 2001 IEEE Symposium on Security and Privacy*, pp.144–155 (2001).
9) Shioya, T., Oyama, Y. and Iwasaki, H.: A Sandbox with Dynamic Policy Based on Execution Contexts of Applications, *Proc. 12th Annual Asian Computing Science Conference* (*ASIAN '07*), pp.297–311 (2007).
10) 品川高廣，河野健二：実行時のフェーズを考慮したセキュリティポリシー記述の簡略化，先進的計算基盤システムシンポジウム（SACSIS2006），pp.495–504 (2006).
11) Wagner, D. and Dean, D.: Intrusion Detection via Static Analysis, *Proc. 2001 IEEE Symposium on Security and Privacy*, pp.156–168 (2001).
12) Wagner, D. and Soto, P.: Mimicry Attacks on Host-Based Intrusion Detection Systems, *Proc. 9th ACM Conference on Computer and Communications Security*, pp.255–264 (2002).

(平成 20 年 7 月 23 日受付)
(平成 20 年 11 月 7 日採録)

**神山　貴幸**（学生会員）
1985 年生．2007 年電気通信大学電気通信学部情報工学科卒業．現在，電気通信大学大学院電気通信学研究科情報工学専攻博士前期課程 2 年．興味はシステムソフトウェア，セキュリティ．

**大山　恵弘**（正会員）
1973 年生．2001 年東京大学大学院理学系研究科情報科学専攻修了．博士（理学）．科学技術振興事業団研究員，東京大学大学院情報理工学系研究科助手を経て，現在，電気通信大学情報工学科准教授．興味はシステムソフトウェア，セキュリティ，プログラミング言語，並列分散処理．