2005－SE－148（4）
2005／5／31

# インストラクションの発行パターンに着目したソフトウェア監視に向けた一検討

金野　晃、中山　雄大、竹下　敦
（株）NTT ドコモ マルチメディア研究所

ソフトウェア動作の異常を監視する技術において，攻撃見逃し率（異常動作を正常動作と判断する確率），および誤検出率（正常動作を異常動作と判断する確率）の低い監視手法について検討する．提案手法は，ソフトウェア動作の最小単位であるインストラクションを監視のレベルとし，ソフトウェアの過去の動作の学習によってソフトウェアの正常な動作をモデル化する．提案手法の有効性確認のために，脆弱性の存在が明らかにされているソフトウェアの挙動データを取得し，攻撃検知と未学習動作の正常異常判断の予備実験を行った．予備実験の結果，提案手法が未学習に伴う誤検出率を低減し，既存技術では検知できない偽装攻撃を検知できることを確認した．

# A Study of Intrusion Detection Method Focusing on Issuing Pattern of Instructions

Akira Kinno, Takehiro Nakayama, and Atsushi Takeshita
Multimedia Labs, NTT DoCoMo, Inc.

This paper proposes a new software intrusion detection method, which achieves low false negative and positive rates. Our method monitors issuing pattern of instructions, which is the smallest unit of software behavior, and makes probability models of software behavior by learning. To confirm the validity of our method, we conducted some pilot studies. For the studies, we got behavior data of software, which are published their vulnerabilities and attack codes. The results of the studies show our method reduces false positives caused by un-learned behaviors, and it has abilities to detect the mimicry attacks, while existing methods cannot detect them.

## 1.　はじめに

PC やワークステーション、サーバ、ルータ、携帯電話、PDA など、すべての計算機は外部もしくは内部からの攻撃にさらされている。特に、携帯電話は、緊急通報に使われるなど、ライフラインとして欠かせない計算機であり、PC などよりも高度な安全性が求められている。代表的な攻撃は、計算機上で実行されているソフトウェアの脆弱性を踏み台にしたものである。攻撃者はソフトウェアの脆弱性を利用した悪意のある実行コード（ウィルスやワームなど）を計算機に送り込み、実行中のプロセスの制御を奪い、当該プロセスの権限を利用して不正操作をおこなう。ソフトウェアの脆弱性を利用した攻撃への対策として，ソフトウェアの動作を監視し侵入者を検知する技術（IDS: intrusion detection system）がある．IDS はソフトウェアの実行を監視し，異常が発生したと判断されるときにはユーザに警告する，ソフトウェアの実行を終了させるなどの処理を行う．IDS が有用であるためには，検知のオーバーヘッドが小さいことと，検知精度が高い（すなわち，正常な実行と異常な実行を区別する精度が高い）ことが重要である．

本研究の目的は，オーバーヘッドが小さく検知精度が高いソフトウェア監視システムの構築であるが、まずは、高度な安全性のために検知精度の向上を優先している。本稿では，IDS の中の特に異常検知システム(Anomaly detection)と呼ばれるシステムに着目する．Anomaly detection はソフトウェアの正常な動作をモデル化し，ソフトウェアの実行がそのモデルに従っているかを検査することによって異常を検知する技術であり，未知の攻撃・異常の検知を目指している。

本稿で提案する手法は，ソフトウェア動作の最小単位で

あるインストラクション（命令コード，オペコードとも呼ばれる）を監視のレベルとし，監視対象ソフトウェアの過去の動作（すなわち，インストラクションの発行パターン）の学習によって正常な動作をモデル化する．攻撃によって制御が奪われようとしているプロセスからは大半，監視対象ソフトウェアの仕様に反するインストラクションの発行パターンが得られるため，インストラクションのレベルで監視することで多くの攻撃を検知することができると考えている。また，提案手法は未学習動作の正常異常の判断精度を向上させると考えている．

本稿では，提案手法の有効性を明らかにするために，脆弱性の存在が明らかにされているソフトウェアの挙動データを取得し，攻撃検知と未学習動作の正常異常判断の性能評価予備実験を行った．既存技術と比較して評価を行った結果，我々の実験においては，提案手法が未学習にともなう誤検出率（正常動作を異常と判断する確率）を低減し，かつ，既存技術では検知できない偽装攻撃を検知できることを確認した．

## 2. 既存の異常検知システム

### 2.1. ソース解析に基づく異常検知システム

ソフトウェアのソースコードを予め取得しておき、ソースコードを静的解析することによってソフトウェアの正常な動作のモデルを作成し、ソフトウェアの実行系列がこのモデルに受理されるどうかを検査することによって、正常動作からの乖離すなわち異常動作を検知するものである[4,7,10]。ソース解析は正常と異常を判別する規則をソースコードから自動生成するので，ソフトウェア開発者やソフトウェア利用者が正常な動作を示す規則[5]を生成する必要はない．また，モデル化された動作に関しては誤検出率がゼロであるという利点がある．

Lam らが提案するシステム[4]は，モデルの生成をコンパイラによって行い，モデルが決定性有限オートマトンで記述されることが特徴である．ソフトウェアのソースコードには条件分岐や関数呼び出しなど非決定性を生む箇所が多数存在するため、ソース解析からは非決定性プッシュダウンオートマトンのクラスでしかモデルの生成ができないが，コンパイル時にシステムコールのラベル付与、および独自システムコールの追加を行うことで非決定性を削減している。また，関数ポインタや longjmp 命令，シグナルは、性質上静的解析によってオートマトンに反映させることが不可能であるとされるが，監視対象ソフトウェアを動作させながら、モデルへ動的に反映させることで対応していることも特徴である．

しかし，このシステムは条件分岐や関数ポインタなど実行コードの制御が移り変わる箇所の検証を考慮しているが，演算結果，引数などデータ領域のモデル化が困難なことから，条件分岐，関数呼び出しがソースコードどおりに動作しているかを検証できていない．例えば，無限ループが起きてしまっても正常と判断してしまう可能性がある．

### 2.2. 振舞い解析に基づく異常検知システム

#### 2.2.1. 方法

監視対象ソフトウェアの過去の動作の学習によって正常な動作を判別する規則を生成し，その規則に従って異常を検査するものである[1][2][5][9]．実際にソフトウェアを動かしたことによって得られたデータを基にモデル化を行うため，学習が十分であるという仮定ならば，ソフトウェアの正常動作をデータ領域も含めてモデル化することが可能である．すなわち，静的解析では実現できなかった条件分岐，関数呼び出しの動作検証を学習によるモデル化は実現できる．このことから，本研究では振舞い解析に基づく異常検知システムに注目している．

Forest らが提案するシステム[1]は、監視対象ソフトウェアが実行中に発行したシステムコールを捕らえ、システムコールのペアの相関を正常パターンとしてデータベースに保存しておき、監視時に正常パターンとの逸脱を検知する。

Warrender らは、Forrest らのシステムを受けて、システムコールシーケンスを辞書登録し、パターンマッチを行うアプローチを提案している[9]。

Sekar らが提案するシステム[5]は，システムコールのプログラムカウンタとあわせて発行順に蓄積したものを学習系列とし，システムコールのプログラムカウンタを節点，システムコールを辺とした非決定性有限オートマトンを学習により生成することが特徴である．プログラムカウンタを節点とすることで，ソースコードに記述されたループや関数再起呼び出しをモデル化することができる．

Gao らが提案するシステム[2]は，Sekar らのシステムを拡張し，スタックに詰まれたリターンアドレスの状況までオートマトンの辺の情報に含めている．

#### 2.2.2. 問題

Sekar ら，Gao らは，学習系列を有限オートマトンにモデル化している．有限オートマトンは過去の検証結果を現在の検証に利用しない場合に効率的なモデルであるが、プログラムカウンタを状態としているため、条件分岐などによって入り線が複数ある節点が存在することになり、条件分岐を正しく検証しようとすると、過去の条件分岐の結果を保持する必要があるため、非効率である。逆に条件分岐を検証しなければ，有限オートマトンの効率性は保たれるが，攻撃によってデータ領域上の値（例えば演算結果，引数）を変更され，本来あってはならないパスを通ったとしても正常と判断してしまうことがある（Impossible Path 問題[2][7]）．以上により，ソフトウェアのモデル化に入り線が複数ある節点をもつオートマトンを選択するべきではない．

また，実際のソフトウェアの動作から発行されるシステムコールからモデルを学習するので，攻撃検知精度が高い監視をするためには、学習を十分に行う必要があるが、データ領域に格納されたデータを含めて試行することや，すべての機能の組み合わせを試行することは一般的に不可能である。そのため、ソフトウェアの動作検証時に、本来ソフトウェアの正常動作であるが未学習であるため、正常動作ではないと判断せざるを得ず、誤検出の原因となる。

また、従来技術は，システムコールの発行パターンをモデルとしているが，システムコールの発行は，ソフトウェアの動作の一部をモデル化しているにすぎない．そのため，攻撃者はバッファオーバーフローなどを用いて，モデルに受理されるようにプログラムカウンタやスタックに詰まれたリターンアドレスを偽装し，システムコールをモデルに受理されるように巧みに発行することができてしまう（偽装攻撃）[2][8]．例えば exec 系システムコールを偽装攻撃によって発行されると，攻撃者は/bin/sh を起動できてしまい，悪意ある実行コードを実行することが可能である．偽装攻撃が可能な理由のひとつとして，システムコール引数の正常な状態をモデル化することが困難であることがあげられる[3]．システムコール引数は，ソフトウェア外部からの入力によってさまざまに変化するため，網羅的な学習が不可能であるからである．そのため，システムコールをモデル化するだけでは，今後攻撃のトレンドとなりうる偽装攻撃を検知することができない．

## 3. 提案手法

### 3.1. モデル化手法

上記課題を解決するために，提案手法はソフトウェア動作の最小単位であるインストラクションの発行パターンをモデル化する．偽装攻撃はバッファオーバーフローなどを用いてプログラムカウンタやリターンアドレスを偽装するためのコードを埋め込む．提案手法はシステムコールよりも詳細なデータであるインストラクションの発行パターンを利用した監視を行うため，偽装攻撃をほぼ不可能にできるモデルを生成できる。モデル化手法の特徴は，監視対象ソフトウェアを複数回試行し，その間に監視対象ソフトウェアが発行するインストラクションを学習系列とし，学習済み正常動作を効率よく確実に検査できる木構造モデルと監視対象ソフトウェア動作の統計的特徴をモデル化することで未学習動作の正常異常を判断する統計モデルを学習により同時に生成することである．図 1 に提案手法の概要図を示す。

我々の木構造モデルは，インストラクションシーケンスを辺とし，節点は情報をもたない(図 2)．学習方法は，学習系列をモデル上でトラバースしていき，モデル上に存在しないインストラクションが学習系列に現れた時点で新しく辺・節点を追加していくというものである．プログラムカウンタなど入り線を複数にする情報を節点に含めないため、過去の条件分岐の結果はモデルの各節点が保持していることとなる。すなわち、木構造モデルは条件分岐を正しく検証することが可能なモデルであるといえる．また，検証時にモデル上を決定的にトラバースできるため，検証処理の時間効率が高い．しかし，木構造モデルのみでは未学習動作をすべて異常動作として判定せざるを得ない．

そこで，提案手法ではインストラクションの発行パターンから木構造モデルを生成すると同時にソフトウェアの動作特徴を統計的にモデル化するアプローチを取った．統計量としては，システムコールによるモデル化においても効果の見られた，学習系列ごとに計測した共起頻度（N-gram）を選択した．N-gram とは言語処理の手法の一つで、ストリーム上の連続した N 個の文字あるいは単語の列であり、文字や単語の共起関係を把握できる手法である。ソフトウェアの正常な動作とは、ソースコードとライブラリ上のバイナリコードでおおむね定義されている。このことから、未学習かつ正常

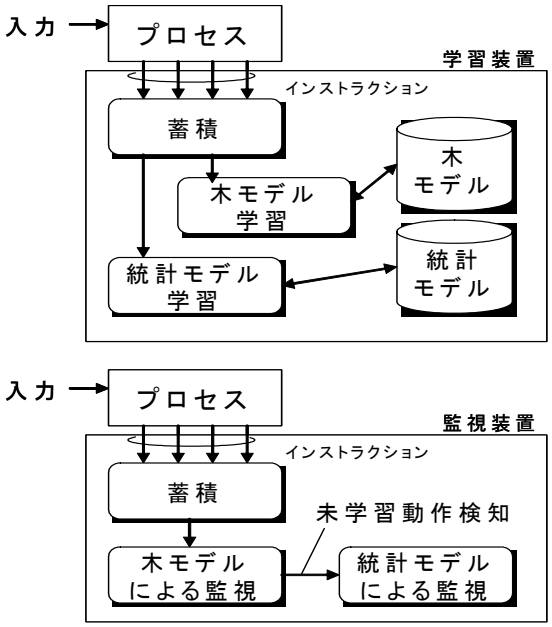


図 2. 提案手法の概要

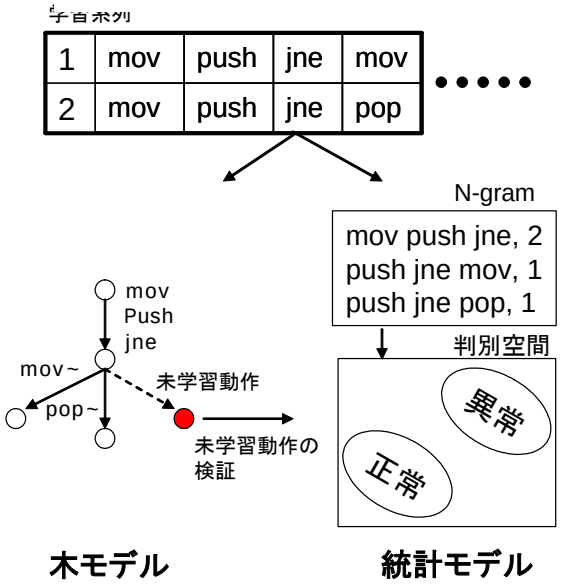


図 1. 木構造モデルと統計モデル

な動作は、同一のソースコード、バイナリコードに存在していたルーチンを利用することとなるため、学習時に得られた統計量と近い値が得られる可能性が高い。一方、未学習かつ異常な動作は、攻撃者が任意の行動をするために、ソフトウェアにはそもそも存在していないコードを埋め込む必要があるため、ソフトウェアの正常動作から得られる共起関係とは異なる統計量が得られる可能性が高い。

統計モデルの学習において，未学習動作の検証を行うために，計測した N-gram をもとに N-gram の各インストラクションシーケンスを説明変数とした判別空間を生成する(図 2). 判別空間上に未学習動作の統計量をプロットすることで，正常，異常の判別を行う．詳しくは 4, 5 章で述べるが，未学習であっても正常動作であれば，判別空間上で正常空間から近い位置に統計量がプロットされることを確認した．

### 3.2. 監視方法

我々の監視方法は，上記のモデル化手法により生成されたソフトウェアの動作モデル（木構造モデルと統計モデル）を用いて 2 段階の監視を行う．ソフトウェアが発行するインストラクションを取得し，1 段目の監視として木構造モデルを用いてインストラクション発行順に動作の追跡を行う．木構造モデル上で逸脱せずにソフトウェアの実行が終了した場合は，正常判定を出力する．逸脱を検知した場合は，検知してからのインストラクションを検証系列とし，2 段目の監視として統計モデルを用いて統計的に正常異常の判断を行う．2 段目の監視では，検証系列から計測した N-gram を判別空間上にプロットし，正常空間との距離を計測して正常異常の判定を行う．このような 2 段階構成をとることによって，正常動作を正常であると効率よく確実に判断できる木構造モデルの利点と，未学習動作をソフトウェアの動作特徴という観点から正常異常の判断を行うことのできる統計モデルの利点を活かすことができる．

## 4. 予備実験

提案手法の有効性を確認するために，一般に脆弱性と攻撃手段が公開されているソフトウェアを用いて予備実験を行った．本章では、実験で利用したソフトウェアを監視対象ソフトウェアと呼ぶ．

### 4.1. インストラクション取得ツール

実験のために，監視対象ソフトウェアが動作中に発行するインストラクションを取得するツール Spbox を実装した．我々の実装では linux カーネルに実装されている実行追跡ツール *ptrace* を利用している．Spbox は，監視対象ソフトウェアを子プロセスとして生成し，*ptrace* のオプション *PTRACE_SINGLESTEP* で子プロセスをステップ実行させながら，*PTRACE_PEEKTEXT* を利用して子プロセスのテキスト領域のデータを取得し，逆アセンブルするよう実装さ

れている．Spbox は，*ptrace* がユーザ空間上の実行を追跡するツールであるため，得られるインストラクションはユーザ空間上のものに限られる．つまり，システムコールなどカーネル空間上で実行されるものについては追跡を行っていない．また本ツールは，*ptrace* の仕様により，監視対象ソフトウェアが生成した子プロセスやライトウェイトプロセス（スレッド）に対しては追跡を行っていない．

### 4.2. 監視対象ソフトウェア

監視対象ソフトウェアは一般に脆弱性と攻撃手段が公開されているソフトウェアである．Spbox の仕様から，これらのソフトウェアは子プロセスを生成しないものを選択した．監視対象ソフトウェアの仕様を表 1 に示す．exim に対しては、バッファオーバーフローを用いて、shell を起動する攻撃を、victim[2]に対しては、バッファオーバーフローを用いてリターンアドレス、プログラムカウンタを偽装するコードを埋め込み、システムコールを発行する偽装攻撃を行う。 これら 2 つの監視対象ソフトウェアは、本来起こりえない振る舞いによって攻撃を受ける。一方、/bin/ls に対しては、巨大な数字の入力への対応がなされていないバグをつき、特定のシステムコールを発行させ続ける DoS 攻撃を行う。すなわち、この監視対象ソフトウェアは、本来起こりうる振る舞いによる攻撃を受ける。

**表 1 監視対象ソフトウェア**

| 名称 | 被攻撃種類 | 学習動作 | 検証動作 |
|---|---|---|---|
| exim 4.4.3 | Buffer over flow | 正常 15 種<br>異常 1 種 | 正常 6 種 |
| /bin/ls 4.1 | Denial of services | 正常 15 種<br>異常 1 種 | 正常 5 種 |
| victim | Buffer over flow & 偽装攻撃 | 正常 1 種<br>異常 1 種 | 正常 1 種<br>異常 1 種 |

インストラクション取得環境は，VMWare 上にゲスト OS として RedHat7.3 を導入し，インストラクション収集ツール Spbox，監視対象ソフトウェアをインストールした．負例を取得するために監視対象ソフトウェアに対して攻撃を行うので，インストール後，正常なシステム環境である状態でスナップショットを取り，システムを正常環境に早急に戻せるよう，正常環境をバックアップした．これにより，正常動作中にインストラクションを収集する際，攻撃による影響のない純粋なデータを取得することができた．

### 4.3. 実験概要

監視対象ソフトウェアを複数回試行し，実行中に発行したインストラクションを実験系列として蓄積した．蓄積した実験系列を，学習時に利用する系列（学習系列）と検証時に利用する系列とに分け，学習系列から木構造モデルおよび統計モデルを生成した．検証時に利用する系列を木構造モデル上で追跡し，逸脱検知後のインストラクションの系列を検証系列として抽出した．抽出した検証系列からインストラクションの N-gram を計測し各システムコールシーケンスの出現確率を算出し，統計モデルである判別空間上にプロットし，線形判別分析を利用して検証系列の分析を行った．

本実験において，判別分析を精度よく行うために，統計モデル学習時に負例を挿入している．負例は監視対象ソフトウェアに対して，攻撃を与えることで得られるインストラクションの発行パターンである．負例を利用する際には，異常動作の特徴を明確に把握するために，得られたインストラクションからまさに異常動作によって発行されたインストラクションを抽出する必要がある．本実験においては，学習時に生成した木構造モデルを用いて，モデルからの逸脱を検知し，検知後からのインストラクションを負例として抽出した．この負例は実験系列として蓄積した．

こうして得られた実験系列を用いて，未学習動作の判断性能および偽装攻撃の検知性能の評価実験を行った．線形判別分析で利用する距離尺度は各群の中心からその標本への分散を考慮したマハラノビス距離を選択した．また，判別空間を形作る説明変数はステップワイズ法を用いて決定した．

### 4.4. 実験結果

#### 4.4.1. 未学習動作の判断性能評価実験

本実験では，監視対象ソフトウェアのうち exim, /bin/ls の実験系列を用いた．実験系列のうち，正常動作 15 種類と，異常動作 1 種類を学習系列とし，正常動作のうち残りの 5～6 種類から検証系列を生成した．ここで、各々の監視対象ソフトウェアの検証系列として選択した動作のうち 1 種類は、今回行った攻撃がついた脆弱性を含んだ機能を利用していることに注意する。N-gram のシーケンス長は 3 である．

判別結果を表 2 に示す．表 2 から exim については正し

く判別されたが，/bin/ls については誤検出（正常動作を異常と判断する）が存在したことがわかる．

**表 2 判別結果**

| 名称 | 検証数 | 正常判定 | 異常判定 | 誤検出率 |
|---|---|---|---|---|
| exim 4.4.3 | 6 | 6 | 0 | 0% |
| /bin/ls 4.1 | 5 | 3 | 2 | 40% |

### 4.4.2. 偽装攻撃への耐性評価実験

本実験では，監視対象ソフトウェアのうち victim に対する実験系列を用いた．実験系列のうち，正常動作 5 種類，異常動作 1 種類を学習系列とし，正常動作で木構造モデルを生成した．正常動作の残りの 1 種類および異常動作の残りの 1 種類を用いて検証系列を生成した．学習系列，検証系列で用いた異常動作は偽装攻撃がされていることに注意する．N-gram のシーケンス長は 3,6 である．このうちシーケンス長 6 はシステムコールシーケンスの列挙において最も優れた検出性能（高い検出率，低い誤検出率）が得られた長さである[6]. また、N-gram の各インストラクションシーケンスの出現確率を算出するための母数は，検証系列長（すなわち，木構造モデルから逸脱したインストラクションから検証系列終了まで）である．

判別分析結果を表 3 に示す．表 3 からシーケンス長 3 では，攻撃見逃し（異常動作を正常と判断してしまう）が見受けられた．一方，シーケンス長 6 では，攻撃を見逃すことなく正しく判別されていることがわかる．このことからも，シーケンス長 6 のほうがシーケンス長 3 よりも効果的であることがわかる．

**表 3 判別結果**

| 名称 | シーケンス長 | 検証数 | 正常判定 | 異常判定 | 攻撃見逃し率 |
|---|---|---|---|---|---|
| Victim | 3 | 正常 1 | 1 | 0 | — |
| | | 異常 1 | 1 | 0 | 100% |
| Victim | 6 | 正常 1 | 1 | 0 | — |
| | | 異常 1 | 0 | 1 | 0% |

## 5. 考察

未学習動作の判断性能評価実験の結果によると，攻撃により本来起こりえない振る舞いを検知する場合，誤検出率は 0%であった．システムコールの発行パターンを用いて N-gram を計測し，判別分析によって検知する場合と比較すると，提案手法のほうが誤検出率を低減できる．なぜならば，システムコールモデルの場合，偽装攻撃を検知すべく負例に偽装攻撃を受けた時の挙動データを用いると，異常動作空間が正常動作空間に内包されてしまい，未学習動作の判断が困難となるためである．すなわち，システムコールモデルの誤検出率は原理的に 50%になる可能性がある．

一方，/bin/ls では誤検出率は 40%であった．この監視対象ソフトウェアは、巨大な数の入力がされることへの対応がされていないために、本来起こりうる振る舞いをし続けてしまい、システムへ DoS 攻撃をしてしまう。 すなわち、前述の本来起こりえない振る舞いを利用して任意の行動をする攻撃を検知することとは問題のクラスが異なる。提案手法はソフトウェアの動作特徴に基づく監視を行っているが，この攻撃のように正例と負例が分離しづらいデータで判別空間を生成した場合は，誤検出が発生してしまう．この攻撃によって攻撃者は任意の行動を行えるわけではないが、システムへ影響を与える異常動作であるので、誤検出率を低減する手法を検討する必要がある．

偽装攻撃はシステムコール発行パターンをモデル化していては検知することのできない攻撃である．もちろんシステムコール引数をシステムコール発行パターンとあわせてモデル化することができれば検知できるが，システムコール引数のモデル化は，外部からの入力も入りうるデータ領域の情報をモデル化することとなるため，有限なモデルを生成することは非常に困難である．一方，提案手法は，ソフトウェア動作の最小単位であるインストラクション発行パターンを用いてモデル化，監視を行う手法である．偽装攻撃耐性実験の結果によると，提案手法が，6-gram の場合，偽装攻撃を検知できたことを確認できる．

また，インストラクションによるモデル化という監視手法において，自然言語処理で広く用いられる N-gram が偽装攻撃のような監視対象ソフトウェアとは特性が異なるコードを挿入した結果なされた異常動作を判別し検知するパラメータとして有効であることが確認できた．今回の実験においては，N-gram のシーケンス長を長くすることで判別精度が高まることを確認した．今後の実験ではシーケンス長をさらに長くし，シーケンス長の最適値を求める必要がある．また，

提案手法は，統計モデルとして N-gram を採用しているが，出現確率を算出するために母集合として検証系列すべてを採用すると，監視対象ソフトウェアが動作している間にランタイムで監視を行うことができない．そこで検証系列をあるタイミングでスライスし，部分列を用いても提案手法は有効であるか評価をする必要がある．検証系列をスライスするタイミングとしては，システムコールが発行される，jmp・call・ret などソフトウェア動作の特性が大きく変わる可能性のあるインストラクションが発行される，単純に固定長で区切るなどが考えられる．特にシステムコール発行でスライスする手法は，システムコールがシステムに影響を与える可能性のある命令であるため，重点的に監視する必要があるという文脈からも，効果的であると考えられる．

また、提案手法の問題として、木構造モデルの空間コストが挙げられる。木構造モデルは学習系列群を可逆に圧縮するデータ構造であるが、学習系列ごとに葉が生成されるため、学習を十分に行うという学習によるソフトウェア動作監視の性質上、空間コストが非常に大きくなることが考えられる。そのため、運用に耐えうるデータ構造を検討する必要がある。

## 6. まとめ

本稿では，ソフトウェアの動作をランタイムで監視する研究を行う上で，ソフトウェア動作の最小単位であるインストラクションを利用したモデル化手法および監視手法を提案した．提案手法は，監視対象ソフトウェアが過去に動作した際に発行したインストラクションを用いて学習により木構造モデルと統計モデルを同時に生成する．監視時において，提案手法は，木構造モデルを利用した監視をまず行い，逸脱を検知すると，検知の時点から発行されるインストラクションを検証系列として統計モデルを利用した監視に変更する．

提案手法の有効性を明らかにするために，脆弱性の存在が明らかにされているソフトウェアの挙動データを取得し，攻撃検知と未学習動作の正常異常判断の性能評価を行った．既存技術と比較して評価を行った結果，提案手法が既存技術では検知できない偽装攻撃を検知し，かつ，未学習にともなう誤検出を低減する可能性を確認した．

今後の課題として，監査手法のさらなる精度向上に向け，N-gram シーケンス長の最適値を導出すること，検証系列をスライスした場合の評価を行う．また，今回の実験で得られた結果の妥当性を明確にするために，大規模なサンプルでの検証実験を行う．また，モデルのサイズ，監査スピードという観点で，インストラクションを監視することによる影響の評価および木構造と統計モデルをスイッチすることによる影響の評価を行う．

## 参考文献


[1] S. Forrest et al, “Intrusion Detection using Sequences of System Calls”, Journal of Computer Security Vol. 6 (1998), pp.151-180.

[2] D.Gao et al, “On Gray-Box Program Tracking for Anomaly Detection,” 13th USENIX Security Symposium, August 2004

[3] C. Kruegel et al, “On the Detection of Anomalous System Call Arguments”, In proc. of 8th European Symposium on Research in Computer Security, Norway, October 2003

[4] L. Lam et al, “Automatic Extraction of Accurate Application-Specific Sandboxing Policy,” In proc. of EUROCOM International Symposium on Recent Advances in Intrusion Detection (RAID), September 2004

[5] G.C.Necula et al, "Proof-carrying code”. In proc. of the 24th ACM SIGPLAN-SIGACT Symposium on Principles and Programming Languages, January 1997.

[6] R. Sekar et al, “A Fast Automaton-based Method for Detecting Anomalous Program Behaviors”, In proc. of the 2001 IEEE Symposium on Security and Privacy, May 2001.

[7] D. Wagner et al, “Intrusion Detection via Static Analysis”, In proc. of the 2001 IEEE Symposium on Security and Privacy, May 2001.

[8] D. Wagner et al, “Mimicry Attacks on Host-based Intrusion Detection Systems”, In proc. of the 9th ACM Conference on Computer and Communications Security, November 2002.

[9] C. Warrender et al, “Detecting Intrusions Using System Calls: Alternative Data Models”, IEEE Symposium on Security and Privacy, May 1999.