社団法人 情報処理学会 研究報告
IPSJ SIG Technical Reports

2006－CSEC－34 (26)
2006／7／20

スタック探索の簡略化による異常検知システムの高速化

鈴木 勝博† 阿部 洋丈†† 加藤 和彦† 金野 晃††† 池部 優佳†††
中山 雄大††† 竹下 敦†††

† 筑波大学 システム情報工学研究科
†† 独立行政法人 科学技術振興機構
††† 株式会社 NTT ドコモ

あらまし 近年，コンピュータはネットワークからシステムへの攻撃といった外部からのさまざまな攻撃にさらされている．我々はこれらの攻撃に対して，学習によってソフトウェアの動作をモデル化し，スタックの情報を利用してソフトウェアの動作を監視するシステムの適用に注目している．ソフトウェアの動作を監視することによって，オーバヘッドが発生しソフトウェアの動作速度が低下する問題がある．なぜならスタック情報の取得に処理時間がかかるためである．この問題に対し我々は，スタックの情報を取得する過程を省略することによる高速化を目指した．
キーワード 異常検知システム，スタックバックトレース

Performance Improvement of Anomaly Detection System by Simplifying Call Stack Inspection

Katsuhiro SUZUKI†, Hirotake ABE††, Kazuhiko KATO†, Akira KINNO†††, Yuka IKEBE†††,
Takehiro NAKAYAMA†††, and Atsushi TAKESHITA†††

† University of Tsukuba.
†† Japan Science & Technology Agency.
††† NTT DoCoMo, Inc.

Abstract Recently, the computer is exposed to various attacks from the outside and the inside of the system. We are studying to the anomaly detection system that observes software behaviors by learning, and using information on the stack. There is a problem that increase the overhead of software by observing the pattern. Because to take time to inspect the stack. We proposed at improve the performance by simplifying the stack inspection for this problem.
Key words Anomaly Detection, Call Stack Inspection

1. はじめに

近年，コンピュータはネットワークからシステムへの攻撃といった外部からのさまざまな攻撃にさらされている．コンピュータへの攻撃として代表的なものに，ソフトウェアの脆弱性を利用したものがある．例えば，境界チェックを行っていないバッファがあるために，バッファサイズを越える大きなデータを送りつけることで，本来ならばバッファとして用いられないメモリ領域にデータが書き込まれてしまう，脆弱性 (バッファオーバフロー) がある．攻撃者はこの脆弱性を悪用して，ソフトウェアに攻撃コードを注入し実行させることができる．攻撃コードとは，正規のソフトウェアにはなかった動作を行わせるプログラムコードのことである．例えば攻撃者がシェルを実行させる攻撃コードを実行すると，サーバソフトウェアが攻撃者に乗っ取られてしまう．

従来より広く用いられているセキュリティ対策として，アンチウイルスソフトウェアが挙げられる．このシステムが用いる基本的な手法は，ウイルスや悪意のある攻撃などのリストを持ち，監視対象となるソフトウェアやソフトウェアが入力するデータに，リストに載っているようなウイルスや悪意のある攻撃が存在していないかを監視する手法である．先の例で言えば，攻撃者がシェルを実行させる攻撃コードをリストに登録しておき，サーバソフトウェアが送受信するデータを監視する．もしサーバソフトウェアに対しリストと一致するデータが送信され

ればそのデータを検知し，管理者に異常を知らせるなどの行動を行う．この手法は新しく攻撃が発見されるたびにリストを最新の状態に更新する必要があり，さらにリストに載っていない未知の攻撃には対処できないという問題がある．

我々はソフトウェアへの攻撃を防御する方法として，異常検知システムを適用することに注目している．例えば先のようにバッファオーバフロー脆弱性を突いて，サーバソフトウェアを乗っ取る攻撃を受けると，攻撃されたソフトウェアは正常時とは異なる動作をする．異常検知システムはソフトウェアの動作を常に監視し，その動作が正常な動作から逸脱していないか検査することで，攻撃による異常動作の傾向を捉え防御する仕組みである．異常検知システムを用いれば，未知の攻撃であっても，プログラムを誤動作させるような攻撃であれば検出できるという特徴がある．しかし異常検知システムには，ソフトウェアを常に監視しながら動作させるために，監視対象ソフトウェアの実行速度が遅くなってしまう問題がある．

本研究では，異常検知システムのオーバヘッドを削減する手法を提案する．我々が対象とするシステムは学習によってモデルを生成し，コールスタック情報から仮想的な関数遷移パスを生成しそれを用いて異常検知を行うシステムである．現状では Feng らの手法 [4] およびその発展である金野らの手法 [6] が存在する．そして前回のシステムコールと，今回のシステムコールから得られたコールスタックの差分から Virtual Path と呼ばれる仮想的な関数遷移のパスを作成し，異常検知システムが保持しているモデルとの比較を行うことで，高精度な異常検知を目指すものである．上記の手法では仮想的な関数遷移パスを作成する際に，スタックの全情報を毎回取得するが，関数遷移パスの作成にはスタックの全情報が不要であることが多い．本研究では，その不要な部分を省略しオーバヘッドを削減する．具体的には，関数遷移パスを作成する際に不要となる箇所を学習によって発見する．不要となる箇所を識別する情報を，終端情報という．異常検知の際に，スタック情報内に終端情報を発見したら，スタック情報の取得を終了させる．

2. 異常検知システム

正常時のソフトウェアの動作を表現したものを，ソフトウェアのモデルと呼び，監視対象のソフトウェアがモデルと逸脱する動作を行った場合に異常と見なす．異常検知システムをモデルの生成方法で大別すると，解析によるものと，学習によるものの二つに分けられる．以下，本研究で着目する学習によるモデルの生成方法について説明する．また異常検知システムが異常検知に用いる情報について述べ，異常検知に用いる情報を取得する際の問題点について述べる．

2.1 解析によるモデルの生成

解析によってモデルの生成では，監視するソフトウェア自体を解析してソフトウェアがどのような振る舞いをする可能性があるかを網羅的に調べる．事前の解析によって網羅的なモデルが完成するため，後に述べる学習のように事前にソフトウェアを実行して動作を観察する期間が必要ないという利点がある．

解析のために必要な情報は，ソフトウェアのソースコードや，ソフトウェアのバイナリなどであり，システムによって異なる．

例えばソフトウェアのソースコードを必要とする例として，Wagner らの手法 [2] が挙げられる．これはソースコードを非決定性プッシュダウンオートマトン (NDPDA) に変換し，モデルとする．

また，ソフトウェアのバイナリコードを必要とする例として，Giffin らの手法 [3] や阿部らの手法 [5] が挙げられる．例えば阿部らの手法 [5] ではバイナリコードを逆アセンブルして，ある関数がどの関数を呼び出すか，といった関数呼び出しの関係を作成しモデルとする．これらの手法はソースコードが入手できないソフトウェアにも適用できるという利点を持っている．

2.2 学習によるモデルの生成

学習による異常検知システムでは，監視するソフトウェアの正常な動作を観察し，モデルとして記録するための学習期間を設ける．学習期間中に発生する動作は全て正常動作でなければならないため，期間中はソフトウェアが攻撃を受けないことが確実にわかっている環境で行う必要がある．

学習によるモデルの生成を行う例としては，システムコール列から N-gram を生成してモデルとし，実行時に発行したシステムコール列との相違を見る Hofmeyr らの手法 [1] や，後述する Virtual Path を作成する Feng らの手法 [4]，金野らの手法 [6] も学習によるモデルの生成を行うシステムである．

2.3 異常検知に用いる情報

ソフトウェアの動作を監視するにあたって，多くの異常検知システムではシステムコール列を監視する方式が用いられる．システムコール列を監視対象とする理由は，攻撃者がシステムコールを全く用いずに，システムに致命的な攻撃を加えることは困難だと考えられるためである．例えば，攻撃者が機密情報の記述されたファイルを盗み出そうとしたとき，必ずそのファイルを開き (このとき open システムコールが使われる)，内容を読みだし (read システムコール)，攻撃者のコンピュータに送る必要がある．これら一連の流れをシステムコールを一切使わずに行うことは困難である．

ソフトウェアが動作する際に，ローカル変数や関数のリターンアドレスを記録する領域をスタックと言い，スタックにプッシュされている情報を，本稿ではスタック情報と呼ぶ．スタック情報の取得方法については後述する．システムコール列による異常検知より精彩な検知手法として，スタック内の情報を利用することが考えられている．例えばシステムコール列のみを監視する手法では，正常なシステムコール列と同様のシステムコールを発行して攻撃する，なりすまし攻撃を防げない．また，意図されていないライブラリの関数を実行させソフトウェアを攻撃する，return into libc 攻撃でもシステムコールを発生させない攻撃が可能であるため防げない．スタック情報を用いれば，どの関数を経由してシステムコールが発行されたか推測できる．これによってなりすまし攻撃のために正規の処理では存在しない順序で関数の呼び出しを行ったり，return into libc 攻撃のために通常なら関数が呼ばないライブラリ関数へ制御が移ったことを検出できる．

2.4 取得時の問題点

スタック情報を利用して異常検知を行うシステムのオーバヘッドは，大まかに言えば3つに分けられる．

（1） 監視対象となるプロセスを停止させるためのオーバヘッド

（2） 監視対象となるプロセスのスタックを調査するためのオーバヘッド

（3） 監視対象の振る舞いがモデルに違反していないか検証するためのオーバヘッド

これら3つのうち，(3)については，各異常検知システムによってアルゴリズムが異なり，共通した最適化手法は存在しない．システム間で共通のオーバヘッドは (1) と (2) である．

2つのうち，(1)については，主にOSの機能に依存する部分が大きい．例えば監視対象となるプロセスのメモリやレジスタを読み出すための時間や，監視対象となるプロセスを停止させるまで，異常検知システムが待機する時間である．これらのコストの改善するには変更を加えた特殊なカーネルを導入する必要があるが，改善する箇所はプラットフォームに強く依存する．一方 (2) については，プラットフォームに依存せず，共通のアルゴリズムを用いることが可能である．我々は (2) を改善することによって高速化を図ることに着目した．

3. 対象となる異常検知システム

本研究が対象とするシステムは，Feng らの手法 [4] や，金野らの手法 [6] のように，学習によるモデルを用いており，かつコールスタック情報から仮想的な関数遷移パスを生成するシステムである．

以下，スタック情報を取得するための処理であるスタックバックトレースの説明を行い，その後，コールスタック情報の推定手法を説明し，仮想的な関数遷移パスを生成する手法を説明する．

3.1 スタック情報の取得

ここでは関数遷移パスの作成に必要なスタック情報の取得処理である，スタックバックトレース処理について説明する．

図1はx86のスタック構造である．スタック情報の構造は，アーキテクチャや関数の呼び出し規約に依存しているため，全てのアーキテクチャが図1の通りとは限らない．図1の状態に至るまでの状況は，以下の通りである．

（1） 関数Aが関数Bを呼び出し

（2） 関数Bが関数Cを呼び出し

（3） 関数Cがシステムコールを発行した

システムコールが発行されると，異常検知システムはこのプロセスがモデルに違反した動作をしていないか検査するために，プロセスの実行を一時停止させる．

バックトレース処理の始めに，システムはまず現在のフレームポインタを格納しているレジスタFPを参照する．FPは以前のフレームポインタの値がプッシュされたスタック内のある地点（図1のスタックフレーム1中の FP_1 を指している．例えば，スタックの情報としてリターンアドレスを用いるとする．図1ではFPの指す地点から，一つだけスタックの底に近い方にリターンアドレスがプッシュされている．この知識を利用し，リターンアドレスを取得することができる．これ以降，以前の FP_n の指す地点から，以前の FP_{n+1} を取得し，付近にプッシュされているリターンアドレスを取得することを繰り返す．これを以前のFPが取得できなくなる地点まで続ける．取得できなくなった地点を「スタックの底」と呼ぶ．

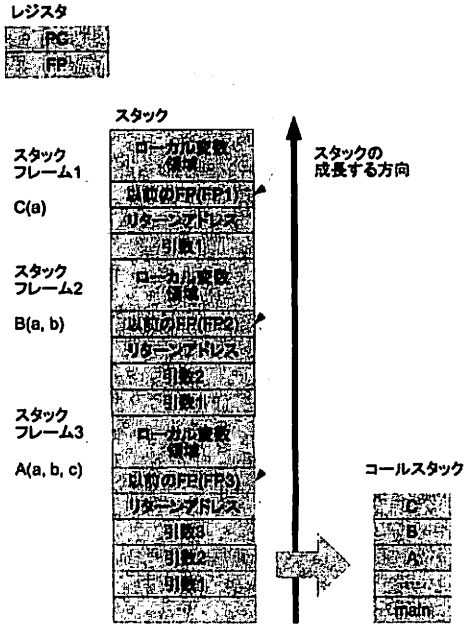

図1 スタックバックトレース

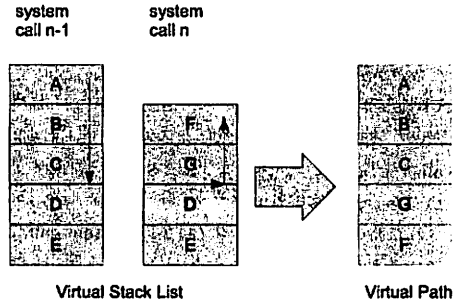

図2 仮想的な関数遷移パスの生成

3.2 仮想コールスタックの構築

ここでは前節のスタックバックトレース処理によって取得したスタック情報から，どのようにして仮想的なコールスタックを生成するかを説明する．

スタックバックトレースで取得したスタック情報のうち，各スタックフレームのリターン情報を用いると，図1の右側にあるようなコールスタックを生成できる．コールスタックとは，現在の関数がどの関数から呼び出されているかを示すものである．図1で言えば，現在の関数Cのリターンアドレスを得るこ

とによって，関数Cが終了すると関数Bに制御を戻すことがわかる．関数Bが終了したときにどの関数に制御を戻すかは関数Bのスタックフレーム（スタックフレーム2）を調べることによって取得できる．これをスタックの底まで繰り返すことで「仮想的な」コールスタックが生成できる．

3.3 仮想的な関数遷移パスの構築手順

得られたコールスタックの集合から関数遷移パスを生成する方法を説明する．関数遷移パスの生成は以下のような手順で行われる．

（1） 仮想的なコールスタックを構築する

（2） 前回のコールスタックと比較し異なる箇所を探す

（3） 異なる箇所から頂上側を連結する

初めに (1) について説明する．異常検知システムは，監視対象のプロセスがシステムコールを発行すると，監視対象を停止させ仮想的なコールスタックを構築する．次の (2) ではシステムは System call$_{n-1}$ と System call$_n$ のコールスタックを底に近い方から順に比較し，異なるリターンアドレスが出現する地点を検索する．最後の (3) では前回システムコールが発行されたときのコールスタックの頂点から，異なるリターンアドレスが出現する位置を終点として，スタックの底に向かう方向（下向きの矢印に相当する）に取る．次に今回のシステムコールで得られた，コールスタックを見る．異なるリターンアドレスが出てくる位置を起点として，底から頂点に向かう方向（上向きの矢印に相当する）に残っている部分を取得する．以上の2つを連結したものが関数遷移パスとなる．

例として図2を用いて関数遷移パスの構築を行う．これは異常検知システムが n 番目に発行されたシステムコール (図2の System call$_n$) を検知し，その時点で監視対象のプロセスが停止した状態である．異常検知システムは前回のシステムコールが発行された時のコールスタック (図2の System call$_{n-1}$) を記録しているものとする．プロセスが停止すると (1) の処理が行われ，コールスタックが構築される．次に (2) の処理が実行される．図2では，底にある関数Eと関数Dのみが共通である．よって (2) の処理では関数Dより上が異なっている，と認識する．(3) では前回のコールスタックを底に行く向き (A→B→C) に取得し，今回のコールスタックを頂上に行く向き (G→F) に取得し，連結する．よって得られる関数遷移パスはA→B→C→G→Fである．

この情報の意味するところは，前回のシステムコールから，今回のシステムコールに至るまで，どのような関数呼び出しが行われてきたか，という推移である．例えば，図2であれば，前回のシステムコールが終了してから，関数Aに制御が戻り，関数Aが終了し，関数Bに制御が戻る．同様に，関数C，関数Dへと戻り，関数Dの実行が進む．ある程度まで実行が進むと，関数Dは関数Gを呼び，関数Gは関数Fを呼び出す．そして関数Fがシステムコールを発行したため，異常検知システムによって停止させられた，という推測ができる．これが「仮想的な」関数遷移のパスである．「仮想的な」と付くのは，この処理ではシステムコールを呼び出さずに終了する関数は捉えられないため，実際にはより多くの関数を実行している可能性があるからである．また前回のコールスタックと今回のコールスタックが共通している部分についても，推測の通りとは限らない．例えば，図2ならば，関数Dは実際には一度終了していて，異なる条件で関数Eから呼ばれたために，関数Gを呼んだだけという可能性があるためである．

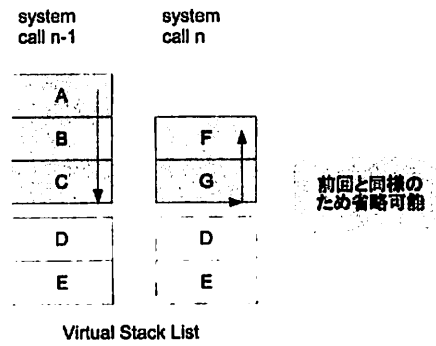

図3 スタックバックトレース処理の省略による高速化

4. 提案手法

関数遷移パスの生成ではシステムコールが発行されるごとに，スタックの底までスタックバックトレース処理を行い，情報を取得していた．しかし，関数遷移パスを生成するにあたって，スタックの底まで取得する必要のない場合がある．提案手法について説明し，提案手法を適用することによって発生する問題について議論を行う．

4.1 高速化手法

図3を例に取ると，前回発行されたシステムコールのコールスタックと，今回発行されたシステムコールのコールスタックの共通点は，関数Dである．もしここで，コールスタック中の関数D以降が高い確立で共通していることがあらかじめ予測できたならば，関数遷移パスの生成に関数D以降の情報は必要ないことがわかる．

本手法ではスタックフレームの取得を省略できる数が多いほど，より効果が大きい．例えば図4のようなループ処理では大抵の場合，非常に似た処理が幾度も繰り返される．するとバックトレース処理の省略が毎回のループに適用でき，提案手法の効果が高い．提案手法ではこれらの知識を事前に学習し，スタックの底までスタックバックトレース処理をせずに，処理を途中で打ち切ることで速度の向上を目指す．

本研究では打ち切る目印となるリターンアドレスを終端情報と呼ぶ．異常検知システムはコールスタックの構築中に終端情報を発見した場合，スタックバックトレース処理を中断し，監視対象アプリケーションの実行を再開させる．

具体的な処理の手続きを図5に示す．本提案手法では，後述する関数遷移パスの推定ミスを削減するため，リターンアドレスとメモリアドレスのセットを保持する出現リストを用いる．もし出現リストにあるリターンアドレスとメモリアドレスのセットが発見できれば，以前と同じ位置に同じリターンアドレ

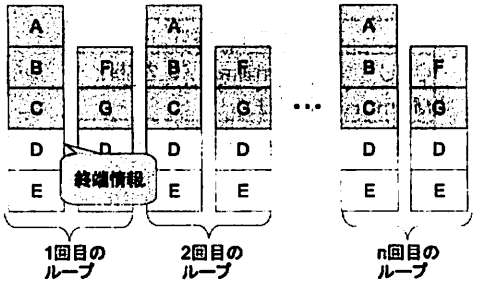

図 4　終端情報が有効に機能する例

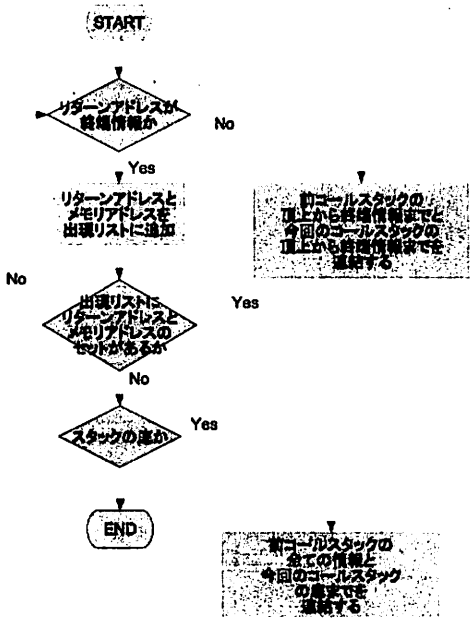

図 5　終端情報によるスタック情報取得手法

スが出現したことがわかるため，その時点でスタック情報の取得を打ち切る．スタックの底に至るまで，出現リスト内のセットを発見できなければ前のコールスタックの全ての情報（前回のシステムコールでスタック情報の取得を打ち切ったならば，打ち切るまでのコールスタックを用いる）を使って関数遷移パスを構成する．

終端情報を利用するためには，バックトレース処理を打ち切る目印となるリターンアドレスをあらかじめ知る必要がある．本研究の対象は，学習によってモデルを生成する異常検知システムであるため，モデルを作成する際の学習と同時に，終端情報の学習を行うことが可能である．終端情報の学習時には，常にスタックの底までスタックバックトレース処理を行い，その際に出現したリターンアドレスを記録する．どのリターンアドレスを目印にして，それ以降のバックトレースを省略すれば，最も効果的なのかを推測し，終端情報となるリターンアドレスを選択する．

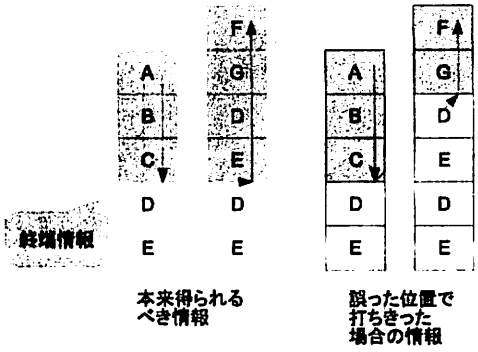

図 6　バックトレース処理の省略により誤った推定を行う例

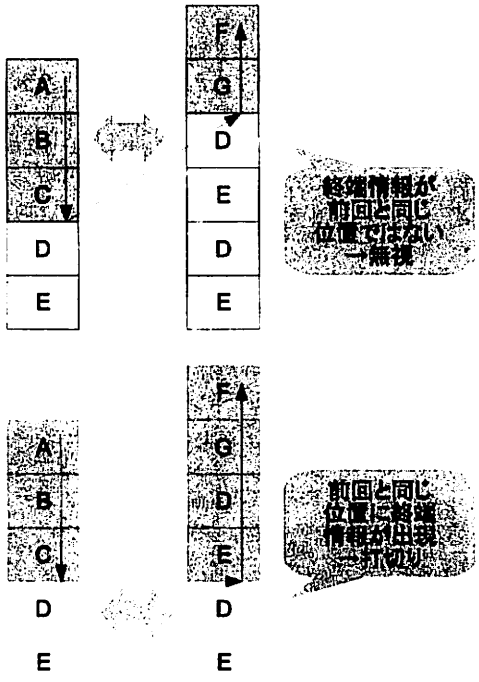

図 7　メモリアドレスのチェックによって推定ミスを防げる例

4.2 議　　論

本提案手法では，関数遷移パスの推定を誤るリスクが存在する．関数遷移パスの推定誤りを未然に防げる場合と，そうでない場合がある．それぞれの場合について説明を行う．さらに関数遷移パスの推定を誤ることによって，異常検知の結果が変化し検知精度が低下する問題について説明を行う．

4.2.1 未然に防げる例

例えば図 6 のような状況を考える．本来スタックバックトレース処理を打ち切る場所は，下から 2 番目にある関数 D のリターンアドレスであるが，下から 4 番目に同様のリターンアドレスが出現している．もしリターンアドレスが一致していることだけで判断してしまうと，本来処理を打ち切る箇所である，下から 2 番目ではなく，より手前の下から 4 番めでスタッ

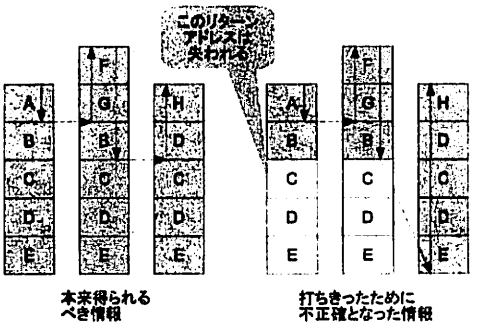

図 8　メモリアドレスのチェックをしても推定ミスする例

クバックトレース処理が打ち切られてしまう．このような現象が起きるため，図 7 に示すように，前回のコールスタックと今回のコールスタックとで，リターンアドレスが格納されているメモリアドレスが一致しているか確認を行う必要がある．

メモリアドレスとコールスタックとの関連性について説明する．スタックはメモリのどこかに確保されるため，スタックへのプッシュやポップはメモリのどこかへの書き込みや読み込みと言える．メモリアドレスとは，コールスタックの底からの位置ではなく，あくまでも，スタック情報がどのアドレスに格納されているかを表すものである．同じメモリアドレスに，異なるスタックフレームの情報が格納されることはない．

この情報を利用すると，下から 2 番目の関数 D と，下から 4 番目の関数 D は必ず別のメモリアドレスに格納される．前回の関数 D が格納されているメモリアドレスと一致するアドレスを持つ方が，同一の位置に居るリターンアドレスである可能性が高いといえる．メモリアドレスの確認により，本来スタックバックトレース処理を打ち切るべきではない位置で，処理が打ち切れてしまうことを防いでいる．

4.2.2　防げない例

メモリアドレスのチェックを行っても，バックトレース処理の打ち切りによって誤った情報が生成される場合がある．例えば，図 8 のような例である．

スタックの底まで取得する場合は，初回は関数 B までが共通のリターンアドレスであり，その次は関数 C までが共通のリターンアドレスであることが容易にわかる．ここで終端情報を関数 B と関数 C としたとき，初回のスタックバックトレース処理で同じ位置に関数 B が発見されるため，その時点で処理は打ち切られる．ここまでは正しい．しかし打ち切った地点より底に近い部分にはどのようなリターンアドレスがあったのかわからなくなる．このためその次に行われるスタックバックトレース処理では，本来ならば同位置に出現する終端情報である関数 C を発見できず，スタックの底まで取得してしまう．その後はスタックの底側から比較が行われるが，一致する部分がないために図 8 のように連結される．

4.2.3　異常検知精度に対する影響

提案手法を適用することで，検査時と学習時とで異なる関数遷移パスを得る可能性がある．これにより本来ならば異常動作ではないのに，異常動作であると報告したり，異常動作であるのに正常動作であるとみなして，見逃したりといった異常検知の精度低下を招く恐れがある．

実験環境	
CPU	Pentium4/3.2GHz
Memory	1GB
OS	Linux 2.6.16.20

表 1　測定に用いた実験環境

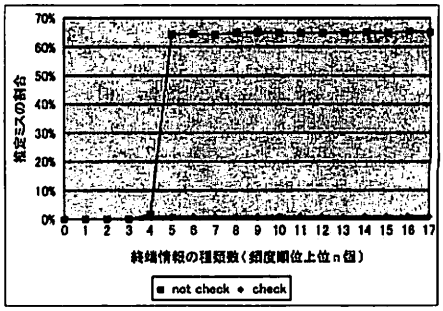

図 9　メモリアドレスのチェックによって防げる推定ミスが多くを占める例

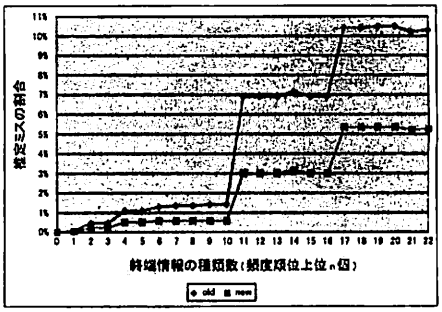

図 10　メモリアドレスのチェックによって防げる推定ミスとそうでないミスが同程度である例

推定を間違う確率は，監視対象となるソフトウェアの性質あるいは使用方法によっても大きく異なるが，少なくともメモリアドレスのチェックをすることによって関数遷移パスの推定ミスが増加することはない．

実際にどの程度の推定ミスが発生するかを測定すると，図 9 や図 10 のような結果となった．実験環境は表 1 である．実験の手順は以下の通りである．

（1）　ソフトウェアを実行させ，全てのスタック情報を使って関数遷移パスを構築する

（2）　終端情報を用いて，同じ条件でソフトウェアを実行させ関数遷移パスを構築する

（3）　用いる終端情報の数を増加させ (2) を繰り返す

実験に用いたソフトウェアの動作は，図 9 が tar で linux-2.6.16.20 のソースコードを展開したときで，図 10 が tar-1.14

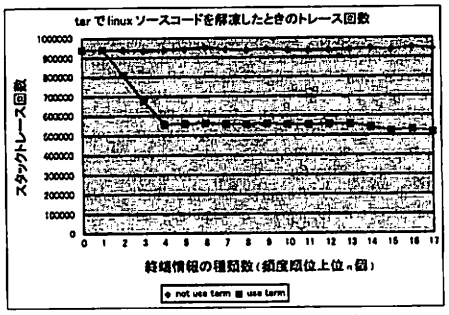

図 11　tar で linux ソースコードを展開した際のトレース回数

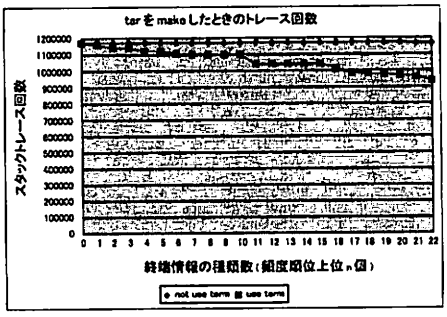

図 12　tar-1.14 を make した際のトレース回数

を make したときである．

図 9 のように，メモリアドレスのチェックによって防げる推定ミスが非常に多発している場合は，メモリアドレスのチェックが非常に高い効果を示す．この例の場合は推定ミスの割合は最大で 0.8%程度である．また図 10 のように，メモリアドレスのチェックによって防げる推定ミスと，そうではない推定ミスが同程度発生している場合もある．この例の場合は推定ミスの割合が最大で 5%程度であるが，関数遷移パスの推定ミスの割合と，異常検知の精度への影響がどのように関係するのかは未調査であるため，この発生率が高いのか低いのかは現時点で論ずることはできない．これは本研究の今後の重要な課題である．

また，実験に用いた 2 つのソフトウェアで全ての可能性を検証しているとは言えず，メモリアドレスのチェックによって防げない推定ミスが多発するような動作をするソフトウェアが存在するかもしれない．これは今後も調査すべき課題の 1 つである．

5. 実験と評価

提案手法を適用することにより，どの程度の高速化が見込めるかを評価した．

5.1 実験の概要

実験の手順は以下に示す通りである．本実験に用いた環境は表 1 の通りである．

（1） 短時間で実行が終わるような小規模な対象を用いて，終端情報の学習を行う．

（2） 学習で得られた終端情報を用いながら，比較的処理時間を要する対象にを監視しながら実行させる．

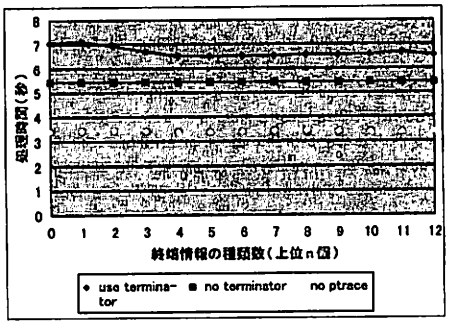

図 13　linux のソースコードを展開した時の処理時間

（3） その結果から終端情報を用いない場合のトレース回数と，比較してどの程度トレース回数を削減できたかを計測する．

（4） 最後に，終端情報を用いる場合と，用いない場合の関数遷移パスを比較し，間違った回数を計測する．

終端情報の選択に用いたアルゴリズムについては，本稿では出現回数の多かったリターンアドレスをリスト化し，その上位からいくつかを選択する単純な手法を用いた．またトレース回数とは，バックトレース処理においてスタックフレームを遡った回数のことである．つまり 5 段のスタックをスタックの底までバックトレースした場合，トレース回数は 5 回となる．

実験では 2 つの性質の異なるソフトウェアを用いた．具体的にいうと，tar と make および gcc である．tar はアーカイブからファイルを読みだし，圧縮されているアーカイブなら展開して，その後ディスクにファイルを書き込むという処理を繰り返し行う．このため，提案手法が最も有効に働くと考えられる例である．make や gcc は tar のように一定の処理を延々と繰り返すソフトウェアではなく，様々な関数が呼び出されるソフトウェアである．このため提案手法があまり有効に働かないと考えられる例である．

5.2 実験結果

実験結果を図 11 と図 12 に示した．

図 11 は linux のソースコードを tar で解凍，展開したときの様子である．初めに mozilla-1.7.13 のインストーラが格納されているアーカイブ（ファイル数 20）を解凍することによって，終端情報の学習を行った．次に linux-2.6.16.20 のソースコードが格納されているアーカイブ（ファイル数 204311）を解凍し，スタックバックトレースの回数がどの程度削減できたかを測定した．tar のように，単純な処理を反復して行うソフトウェアでは，大幅に削減可能で，スタックバックトレースの回数を約 6 割にまで削減できた．

図 12 は tar を make したときの様子である．初めに hello-2.1.1 という小さなプログラムを make することで，終端情報の学習を行った．次に tar-1.14 の make を行い，スタックバックトレースの削減回数を測定した．make および gcc のように，複雑な処理を行うソフトウェアでは提案手法があまり有効ではないものの，約 2 割の削減ができる．

全体の処理時間に対する効果を図 13 に示す．これは tar で linux-2.6.16.20 を展開したときの処理時間である．I/O の待ち時間による影響をできるだけ排除するため，RAM 上に作成した領域に対して展開を行った．比較対象として節 2.4 の (1) のオーバヘッドのみの処理時間 (no terminator) と，監視を行わないときの処理時間 (no ptrace) を示した．終端情報の利用により，(2) のオーバヘッドが約 4 割削減されていることがわかる．

6. まとめと今後の課題

コールスタックにどのようなリターンアドレスが出現するか事前に学習した情報（終端情報）を利用し，関数遷移パスを生成する際に行われるスタックバックトレース処理を省略することにより，異常検知システムのオーバヘッドを低減する手法を提案した．

提案手法が対象とするのは，学習によってモデルを生成し，コールスタックから関数遷移パスを生成して利用する異常検知システムである．

今後の課題として，より効果的な終端情報の選択方法を考案，評価すると共に，提案手法が異常検知の精度に与える影響を数値的に評価したい．

文　献

[1] Hofmeyr, S.A., Forrest, S., Somayaji, A.: Intrusion Detection using Sequences of System Calls. Journal of Computer Security Vol.6, No.3. pp.151-180 (1998)

[2] Wagner, D., Dean, D.: Intrusion Detection via Static Analysis. In Proceedings of the 2001 IEEE Symposium on Security and Privacy, Oakland. pp.156-168 (2001)

[3] Giffin, J.T., Jha, S. and Miller, B.P.: Detecting manipulated remote call streams. In Proceedings of the 11th USENIX Security Symposium. pp.61-79 (2002)

[4] Feng, H.H., Kolesnikov, O., Fogla, P., Lee, W. and Gong, W.: Anomaly Detection Using Call Stack Information. In Proceedings of The 2003 IEEE Symposium on Security and Privacy. pp.62-75 (2003)

[5] 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：コンピューティングシステム，Vol.45, No.SIG 3 (ACS 5), pp.11-20 (2004)

[6] 金野晃，池部優佳，竹下敦，中山雄大，加藤和彦，阿部洋丈，鈴木勝博．携帯端末向けソフトウェア異常検知技術．情報処理学会：システムソフトウェアとオペレーティング・システム研究会．pp.1-8 (2006)