2006－OS－102（1）
2006／5／12

携帯端末向けソフトウェア異常検知技術

金野　晃[†1]，池部　優佳[†1]，中山　雄大[†1]，竹下　敦[†1]，
鈴木　勝博[†2]，阿部　洋丈[†3]，加藤　和彦[†2]
†1（株）NTT ドコモ マルチメディア研究所
†2 筑波大学
†3 科学技術振興機構

PC やサーバ，携帯電話など，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．われわれはこの中でも携帯電話を攻撃から守るために，スタック情報を用いたソフトウェアの動作検証方法に注目した．携帯電話など処理能力の低い計算機において, 動作を検証するシステムを搭載するためには, 限られた資源での処理の高速化, 使用メモリ量の抑制が重要となる．そこで，われわれは低速なメモリアクセスを伴うスタック探索を簡略化し，オーバーヘッドを低減することを試みた．

Anomaly Detection Technique for Mobile Terminals

Akira Kinno[†1], Yuka Ikebe[†1], Takehiro Nakayama[†1], Atsushi Takeshita[†1],
Katsuhiro Suzuki[†2], Hirotake Abe[†3], and Kazuhiko Kato[†2]
†1 Multimedia Labs, NTT DoCoMo, Inc.
†2 Tsukuba University
†3 Japan Science and Technology Agency

We propose an anomaly detection technique, which achieves low overhead. It monitors issuing pattern of system and function calls, and makes probability models of software behavior by learning. To apply this technique to mobile terminals, it is necessary to speed up the process and reduce the memory size because of limited resources of mobile terminals. In this study, we propose the method for reducing overhead of the anomaly detection software by simplifying the stuck trace. As a result of preliminary experiments, it is recognized that our method can be effective.

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， 我々は，異常検知システム(Anomaly detection)と呼ばれる手法に着目する．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監視することによって異常を検知する手法であり，未知の攻撃・異常の検知を目指している．

本研究の目的は，Anomaly detection を，携帯電話をはじめとする，モバイル端末に搭載し運用に耐えうるシステムを構成することにある．本研究では，モバイル端末の特性を鑑みて，監視アルゴリズムの要求条件を，(1) 検知精度が高い，(2) 監視の処理オーバーヘッド（時間・空間）が小さい，および (3) 消費電力が低いこととし，監視アルゴリズムを検討する．なお，本稿では，要求条件(2)に着目する．

本稿で提案する手法は，ソフトウェアの発行するシステムコールとその時点でプロセススタックに積まれたリターンアドレスとの共起関係に着目し，監視対象ソフトウェアの過去の動作（すなわち，システムコールとリターンアドレスの発行パターン）の学習によって正常な動作をモデル化する．共起関係のモデル化を行う際，統計手法としてモデルサイズを小さく抑えることが可能な相関規則を採用した．さらに，モデル化の際の知識を利用して，監視時において時間オーバーヘッドの大きなリターンアドレスの取得を簡略化する．このように Anomaly detection を構成することで，高い検知精度を維持しつつ低オーバーヘッドで異常挙動を検知できると考える．

本稿では，提案手法の有効性を明らかにするために，まず予備実験として，組み込み機器上に，我々の提案したモデル化および監視アルゴリズムを実装し，それぞれにかかる処理コスト評価を行った．さらに，スタック探索の簡略化を実装し，処理コストにおける提案手法の効果を測定した．実験結果より，提案手法は処理時間削減に有効であり，処理コスト低減に有効であることを確認した．

2. 既存の異常検知システム

現在用いられている異常検知システムには以下に示す二つの類型がある. ここではそれぞれの概要, 問題点を説明する.

2.1. ソース解析に基づく異常検知システム

ソース解析に基づく異常検知システムは，ソフトウェアのソースコード（あるいはバイナリコード）を予め取得しておき，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[1][3][7][9]. 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[8]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．

しかし，このシステムは，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．また，通常の実行では起こり得ない実行パス(Impossible Path) によって false negative が起きる可能性がある．

2.2. 振舞い解析に基づく異常検知システム

振舞い解析に基づく異常検知システムは，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][5][8][11]. 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．

Forrest 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．

Forrest らのシステムのように，システムコールの発行パターンをモデル化する手法は多数提案されているが，脆弱である．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，攻撃者はモデルに受理されるようにシステムコールを巧みに発行することが出来てしまう（偽装攻撃）[5][10].

そこで，システムコール発行パターンに加え，システムコール発行時の他のパラメータをモデル化することで，偽装攻撃を困難にする手法が提案されている[3][5][8].

Sekar らが提案するシステム[8]は，システムコールに加え，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[5]は, Sekar らのシステムを拡張し，スタックに積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．

Feng らが提案するシステム[3]は，システムコールの正当性を検証するために，コールスタックの状況（スタックに積まれたリターンアドレスの列）を利用する．プログラム実

行中にシステムコール発生時のコールスタックの状況を取得し，システムコール発生時のプログラムカウンタとともに記録した Virtual Stack List を生成し，また，現在の Virtual Stack List(Sn-1)と一つ前の Virtual Stack List(Sn)との差分情報，すなわち比較対象のコールスタックの状況のスタック最下位より比較検証を順次行い異なるリターンアドレスを検出してから以降のリターンアドレスの列（Virtual Path）を生成する（Fig. 1 参照）．生成された Virtual Stack List と Virtual Path からそれぞれハッシュテーブルを生成し，そのテーブルをソフトウェアのモデルとして利用する．ソフトウェアの動作を検証する際には，ソフトウェア実行中に，Virtual Stack List と Virtual Path を生成し，モデルであるハッシュテーブルとのマッチングを行い，合致していればシステムコール要求に対し許可をし，合致しなければ，異常であると判定する．

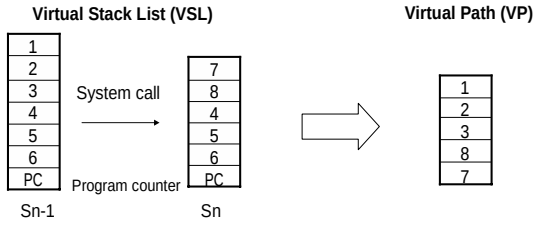

Fig. 1　Virtual Path 生成

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[5][9]）．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などからモデルを学習するので，検知精度の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．Feng らの提案するシステムでは，Impossible Path 問題が起こりうるオートマトンを利用していないが，ハッシュテーブルによる文字列マッチングを行うため，学習時に得られた文字列と一致しなければ異常と判断してしまうため，誤検出の問題がある．

また，Gao ら，Feng らのシステムでは，システムコール発行時に，コールスタックの状況を取得するが，スタック領域は通常メモリに用意され，スタックフレームにはリターンアドレス以外にも引数などの情報が含まれることから，スタックからリターンアドレスを取得するには低速なメモリアクセスを頻繁に行わなければならず時間オーバーヘッドが高い．

3.　提案手法

3.1. モデル化手法

未学習動作に対する誤検出を改善するために，提案手法は Feng らの手法を拡張し，ソフトウェアの発行するシステムコールとその時点でスタックに積まれたリターンアドレスとの共起関係に着目し，監視対象ソフトウェアの過去の動作（すなわち，システムコールとリターンアドレスの発行パターン）の学習によって正常な動作をモデル化する．正常動作を一意に特定してしまう Feng らの手法とは違い，提案手法は，正常動作を統計的にモデル化するアプローチを取る．Feng らの手法では，システムコールとスタックに積まれたリターンアドレスをモデル化することで異常挙動を高い精度で検知することに成功している．このことから，システムコールとリターンアドレスは強い共起関係にあると考えられる．すなわち，未学習かつ正常な挙動は，学習時に得られた特徴と近い特徴が得られる可能性が高いと考えられる．一方，未学習かつ異常な挙動は，学習時に得られた特徴とは異なる特徴が得られる可能性が高い．統計量としては，共起関係を表す統計モデルであり，かつモデルサイズを小さく抑えることが可能な相関規則を採用した．

相関規則とは，ある事象が発生すると別の事象が発生するといったような，同時性や関係性が強い事象の組み合わせ，あるいはそうした強い事象間の関係のことであり，例えば，

アイテムの集合(商品の集合)とトランザクション(商品を購入した顧客のデータベース)の中から，“ ある顧客が商品 X を購入するならば商品 Y も同時に購入する ”というルールを抽出するための手法である．相関規則はマーケティングなどに使われ，商品の配置，仕入れ目安などに使われる手法である．この例では“ 商品 X を買う ”という行為が前例であり，これに対して“ 商品 Y を買う ”という行為が結果となり，前例と結果の関係を相関規則として表す．また，前例と結果を同時に満たすトランザクションが全トランザクションに占める割合，すなわち規則そのものの出現率，のことを support と呼び，前例と結果をともに含むトランザクション件数を全トランザクション件数で除算した値となる[1]．一般に，support が低い規則は，めったに起こらない事象関係であるため，あまり利用されない．そのため，相関規則を導出する際は，support の最小値（minimum support）を指定し，それを満たさない相関規則は除外されることが多い．我々の提案手法においても，モデルサイズを小さく抑えるために minimum support を指定している．

提案手法では，Virtual Path を生成し，Virtual Path 上にあるリターンアドレスを前例，システムコールを結果とした相関規則を生成する．モデルのデータ構造については，3.3. 章で述べる．

3.2. 監視方法

我々の監視方法は，ソフトウェア実行中に Virtual Path を生成し，当該 Virtual Path 上のリターンアドレス，システムコール識別子の共起関係が，モデルである相関規則にマッチしているかを判定する．ここで，異常挙動を，前例はマッチしているが結果は異なる Virtual Path が生成される挙動である，と定義する．提案手法では，システムコールの発行が正当であるかを監視するので，システムコールに至るまでのリターンアドレスがモデルと一致するにもかかわらず，発行されるシステムコールがモデルと異なる場合，異常動作とみなすことができるので，ペナルティを与える．ここでペナルティには Anomaly Score という値を用いる． Anomaly Score とは，異常の度合いを示す値であり，Zero-frequency という確率値の逆数として我々は定義している．Zero-frequency とは前例に伴う新しいイベントの起こる確率であり，結果の種類数を前例が現れた回数で除算した値で近似できることがわかっている[12]．この値が低いほど，未学習な動作が起こりにくいといえるため，ここで異常を検知すると多くのペナルティを与えるべきものであることを示す．このようにして計算された Anomaly Score を元に，異常を判断する．

3.3. モデルのデータ構造

監視における処理コストを軽減するために，モデルのデータ構造は，前例と結果のマッチングと異常値計算が容易に行えることが望ましい．相関規則においてはトランザクション中のアイテムの並び順序は意識しないため，前例と結果のマッチングは組み合わせの計算となり時間オーバーヘッドが高い．そこで我々は，データ構造として FP-Tree 構造を採用した[5]．FP-Tree 構造は相関規則を低オーバーヘッドで生成するためのデータ構造であるが，その特徴は前例と結果のマッチングおよび異常値の計算においても活かされる．

FP-Tree 構造は，頻出アイテムだけが含まれる一種の Trie であり，頻出アイテムがひとつのノードに集約されるよう，根ノードから葉ノードに向けて頻度順にアイテムが並ぶよう構成されるデータ構造である(Fig. 2). 各ノードはアイテム名と頻度で構成されている. FP-Tree を構築するためには，(1) 全トランザクションで各アイテムの頻度を計測し，頻度順のランクリストを生成する，ここで頻度にしきい値をもうけ，ある値以上の頻度を持つアイテムのみ採用することもできる(minimum support). (2) 各トランザクションのアイテムをランクリストに従って頻度順に並べ替える，(3) minimum support 値よりも頻度が少ないアイテムはトランザクションから除外する，(4) 木構造を構築する，という 4 ステップで実現できる．木構造の構築は，初期においてルートノードを生成し現在のノードをルートノードにしておく，各トランザクションでアイテムを検知した順に子ノード（頻度は 1 に設定）にし，パスでつなぎ，現在のノードをその子ノードに移す．検知したアイテムが子ノードに存在していたらノードは生成せずに該当する子ノードの頻度に 1 を加算し，現在のノードをその子ノードに移す作業を，全トランザクション終了まで繰り返すことで実現できる．

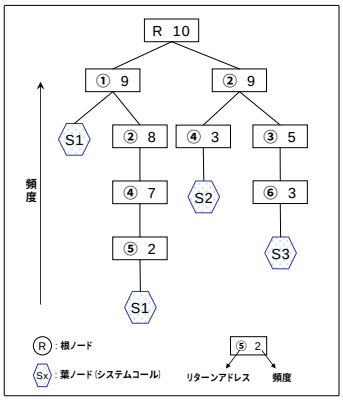

Fig. 2　FP-tree 構造

提案手法では，トランザクションを Virtual Path とし，Virtual Path 上のリターンアドレスをアイテムとして FP-Tree を構築する．さらに，提案手法では，Virtual Path 上のシステムコールを，葉ノードとなるよう，各トランザクションにおける木構造構築で処理する．後述するが，システムコールを葉ノードにすることで，効率よく異常値を計算することができるようになる．提案手法は，こうして得られた FP-Tree と，FP-Tree 構築(1)ステップで生成したランクリストとをモデルとする．

監視の際には，FP-Tree を利用して Virtual Path の検証を行う．すなわち，Virtual Path 上のリターンアドレスの組み合わせが，FP-Tree 上にパスとして存在するか検証する．まず，Virtual Path において，リターンアドレスをランクリストに従って頻度順に並べ替える．FP-Tree はルートノードに近いほど頻度の高いノードが存在するので，リターンアドレスを頻度順に並べ替えることで，FP-Tree 内サーチをルートノードから幅優先探索することができるので，効率がよい．サーチを行っている時に，現在の検証対象であるリターンアドレスが子ノードに存在しない場合，そこでサーチを終了し，現在のノードに属するすべての葉ノード（すなわち，システムコール）を収集する．そして，Virtual Path 上のシステムコールとマッチする葉ノードが存在するかを検証する．そのような葉ノードが存在すれば，Virtual Path は正常と判断し，存在しなければ，異常と判断する（異常挙動の定義は 3.2.章を参照）．異常の場合は異常値計算を行う．

3.4. スタックフレーム探索の簡略化

監視における処理コストを軽減するために，提案手法は，低速なメモリアクセスを伴うスタックフレーム探索を簡略化する．提案手法において，Virtual Path はモデル化，監視両過程において生成される．そこで，モデル化時に生成した Virtual Path の情報から，監視時のスタック探索を簡略化する指針を得る．ソフトウェア実行において，現在のスタック情報と一つ前のスタック情報を比較すると，関数の中で関数を呼び出す際や，ループの状況下では，前後のスタックでリターンアドレスとして等しい情報を持ち，差分とならないものが多数存在することが考えられる．すなわち，ソフトウェア監視時にこのリターンアドレスをすべて取得することは冗長である．そこで提案手法は，モデル化時に，スタック探索を中断するべき箇所であるリターンアドレス（終端情報）を抽出し，監視の際にスタックフレーム探索において終端情報を見つけたら探索を中断する．本稿では終端情報をスタック中に出現する頻度が高いリターンアドレスと定義した．

4.　予備実験

4.1. 実験概要

4.1.1.　処理時間測定

モデル化，監視に必要な処理時間を測定するために，以下の手順で実験を行った．本実験は，Linux を搭載することができる組み込み機器としてアットマークテクノ社製の Armadillo-9 を用いた．

1. x86 上で，脆弱性および攻撃コードが明らかになっているソフトウェアである“ exim ”，“ ls ”を動かし，スタックをトレースし，VP リストを得た（動作の詳細を Table 1 に示す）
2. x86 上で，VP リストを入力とし，モデルを出力するモデル化プログラムと，VP リストとモデルを入力とし，VP リストを監視し，異常か正常かの判断を出力する監視プログラムの二つのプログラムを作成した
3. 2 のプログラムを ARM 用にポーティングし，Armadillo-9 上に実装した
4. 1 で得られた VP リストの一部（exim; 正常動作 21 種のうち 9 動作，ls; 正常動作 20 種のうち 9 動作）を入力データとし，Armadillo-9 上でモデル化を行った（minimum support=1～10[items]）
5. 4 で得られたモデルを用いて，1 で得られた VP リストの監視を行った（minimum support を=1～

10[items])

6. 4,5 でそれぞれにかかった時間を計測した

Table 1 監視対象ソフトウェア

名称	動作データ	
	正常	異常
exim 4.4.3	21 種	1 種
/bin/ls 4.1	20 種	1 種

4.1.2. スタック探索簡略化

モデル化時に終端情報を記録し, 監視時には終端情報を見つけ次第スタック探索を中断するというプログラムを実装し, スタック探索の処理時間の変化を調べた.

exim はライブラリが複雑であり, ls は環境による変化が大きいと考えられるため, 本実験では tar と make & g++を用いて評価を行った. tar は同じ関数が非常に長い間実行されるプログラムであるが, make と g++は同じ関数が長い間実行されることはあまりなく, 様々な関数が呼び出されるプログラムである.

4.2. 実験結果

4.2.1. 処理時間測定

モデル化に要した処理時間を minimum support 別に Table 2 に示す.

Table 2 モデル化処理時間

minimum support	処理時間 (ms)	
	exim	ls
1	479	122
2	474	120
3	450	118
4	442	116
5	422	115
6	421	115
7	410	115
8	413	115
9	395	114
10	391	113

監視対象動作を 3 つ選び, 監視に要した処理時間を Table 3,Table 4 に示す. ここで, 1,2 は正常動作, 3 は異常動作であり, 1 はモデル化の際に使用したデータである.

Table 3 監視処理時間 (exim)

minimum support	処理時間 (ms)		
	1	2	3
1	94	118	323
2	91	114	320
3	88	112	312
4	87	107	305
5	82	108	301
6	81	108	298
7	89	113	303
8	88	111	300
9	83	109	292
10	82	108	281

Table 4 監視処理時間 (ls)

minimum support	処理時間 (ms)		
	1	2	3
1	83	83	99
2	83	81	98
3	80	81	96
4	81	81	97
5	81	81	92
6	79	79	89
7	80	78	89
8	78	79	85
9	74	70	77
10	55	58	74

4.2.2. スタック探索簡略化

gcc-3.3.5.tar を tar で展開した動作と, make と g++を用いて小規模なプログラムをビルドする動作をスタック探索したときの時間を測定した. (Fig. 3,Fig. 4).

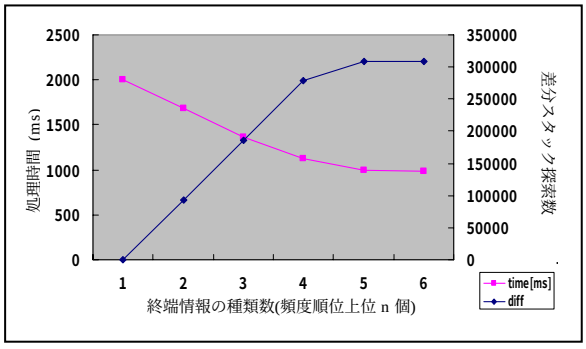

Fig. 3 終端情報を用いた効果(tar)

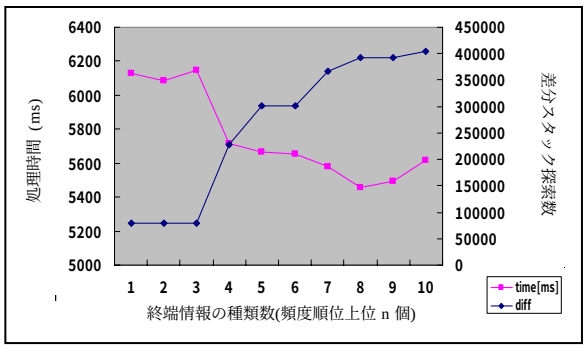

Fig. 4　終端情報を用いた効果(g++)

5.　考察

・処理時間測定

Table 2~Table 4 より，監視対象ごとに比較すると，処理時間は監視対象動作のシステムコール発行数に影響されることがわかり，また minimum support の値が大きいほど，モデル化動作，監視動作とも，処理時間は短くなることが分かる．これは，minimum support が小さいほど，多くの相関規則を採用することになるので，処理時間がかかるためと考えられる. しかし, スタック探索簡略化の実験結果によると，監視処理にかかる時間とスタック探索にかかる時間とを比較すると，監視処理に関する minimum support による時間の差は無視できるといっていい．ただし，実験で用いた監視対象ソフトウェアが異なるため，評価する必要がある．この無視できるということから，検知精度を考慮し，minimum support 値は小さくてもよいといえる．

しかし，minimum support が小さいと，同じ理由からモデルサイズが大きくなることが考えられる．そこで，実際にモデル化を行った際のモデルのファイルサイズを求め，minimum support との関係を調べた（Fig. 5）

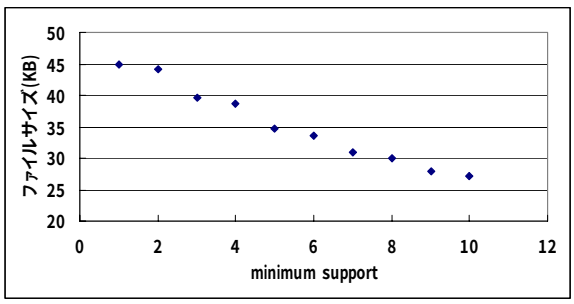

Fig. 5　ファイルサイズと minimum support の関係

これよりファイルサイズは minimum support 比例して小さくなることが確認できる．本システムは，携帯電話という限られた資源に搭載するものであるため，モデルサイズはより小さいほうが有利である．そのため，この考察からは minimum support は大きいほうが有利であると言える．

しかし，minimum support を大きくすることは，相関規則の数を減らすことになるため，モデルに当てはまらない挙動が増えることになる．つまり，精度の悪化を招くと考えられる．精度の評価を行い，モデルサイズ，精度のバランスの取れた minimum support 値を見つける必要がある．

また，監視プログラムにおいて，監視する VP リスト中のリターンアドレスの数と処理時間の関係を調べた．その結果を Fig. 6 に示す. minimum support 値は 1,5,10 で評価した.

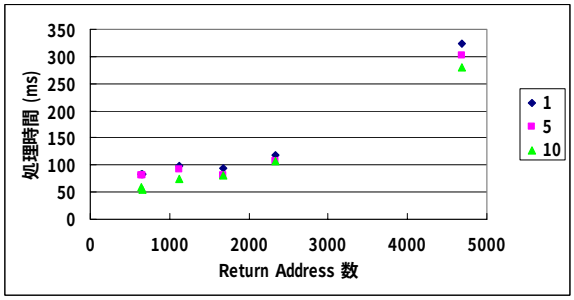

Fig. 6　処理時間と Return Address 数の関係

これより，VP リスト中のリターンアドレス数が多くなるほど，処理時間が大きくなることが確認できる．また，この図を用いることにより，リターンアドレス数からおおよその処理時間を予測することが可能となった.

・スタック探索簡略化

Fig. 3 より，探索するスタックの数は大幅に減り，スタック探索にかかる時間も終端情報を 6 個用いた場合，50%に削減できていることが確認できる．終端情報を用いた方法は，tar のような同じ関数が非常に長い間実行されるプログラムに対しては非常に有効である.

また，Fig. 4 より，g++では 20%の削減ができていることがわかる．g++のように様々な関数が平均的に実行されるプログラムでは，tar に適用したときのような効果はないものの多少の速度改善が見られる.

さらに，どちらの図からも，スタックするフレーム数が減少していることから，モデルサイズも削減できていることがわかり，省メモリ化にも貢献しているといえる.

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，ソフトウェア動作に用いられるリターンアドレスを利用した監視手法のオーバーヘッド低減方法を提案した．我々のソフトウェア異常検知手法は，モデル化において，監視対象ソフトウェアが過去に動作した際に発行したシステムコールと，その時点でスタックに積まれたリターンアドレスの共起関係から相関規則を生成する．また，監視時において，提案手法は，スタックを探索しリターンアドレスを取得する．このスタック探索は低速のメモリアクセスを含むため，オーバーヘッドの増大が懸念される．そこで，モデル化時に見出した終端情報を用いることにより，監視時のスタック探索を簡略化し，オーバーヘッドを低減することを試みた．予備実験として，脆弱性を含むソフトウェア 2 種（exim, ls）の動作のモデル化と監視処理に要する時間を，組み込み機器上で測定した．これより，minimum support が大きいほどファイルサイズは小さく，処理時間も短いことがわかった．また，我々の予備実験においては，監視にかかる処理時間はスタック探索と比較すると無視できるほどであることがわかった．minimum support を決定する戦略として，モデルサイズと検知精度を優先的に検討すべきであるといえる．minimum support はむやみに大きくしてはならない．なぜなら，大きくすればするほど，精度に悪影響を及ぼすことが考えられるからである．また，処理時間をリターンアドレス数と比較することにより，VP リストのリターンアドレスの数から，監視処理に要する時間の目安を得ることが可能となった．また，終端情報を用いたスタック探索の実装を行い，評価した．これより，対象プログラムにより，効果はさまざまであるが，今回の実験では終端情報を用いることにより，20%,50%の処理時間短縮に成功した．

今後，終端情報を用いた手法の精度への影響を調べ，終端情報の定義を最適化していく．

参考文献

[1] 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. SIG3 (ACS 5), pp. 11-20, 2004 年 5 月.

[2] Rakesh Agrawal, “Mining Association Rules between Sets of Items in Large Database.”, ACM SIGMOD Conference, 1993

[3] H. H. Feng, et al, “Anomaly Detection Using Call Stack Information”, In Proceedings of The 2003 IEEE Symposium on Security and Privacy, 2003

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

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

[6] J. Han, H. Pei, and Y. Yin. “Mining Frequent Patterns without Candidate Generation. In: Proc. Conf. on the Management of Data (SIGMOD'00, Dallas, TX)”, ACM Press, New York, NY, USA 2000.

[7] 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

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

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

[10] 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.

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

[12] I. Witten and T. Bell, “The zero-frequency problem: estimating the probabilities of novel events in adaptive text compression”, IEEE Trans. On Information Theory, 1991