静的解析に基づく侵入検知システムの最適化

阿部洋丈[†1] 大山恵弘[†2]
岡瑞起[†3] 加藤和彦[†4,†5]

プログラムの脆弱性を悪用してプログラムがたどる制御フローをプログラマの意図に反するものに変え，プログラムに異常な動作をさせる攻撃が深刻な問題となっている．我々はその攻撃による被害を防ぐ手段として，プログラムがたどる制御フローの異常を検知する侵入検知システムを開発した．そのシステムは，プログラムがたどる制御フローがバイナリコードが規定する制御フローの規則に従っていることを検査しながらプログラムを実行する．検査はシステムコール呼び出し時のスタックを調べることによって行われる．スタック内の制御情報の移り変わりが，バイナリコードが規定する制御フローの規則に従っていないとき，異常が発生したと判断する．我々のシステムはスタックの情報を利用するので，システムコールの情報だけを利用する既存の手法よりも制御の場所を高い精度で把握できる．それはオーバヘッドの縮小と検出精度の向上をもたらす．加えて，我々のシステムは，以前に検査を実行したときに得られた情報を蓄積することで，検査時のオーバヘッドを削減する．我々は実験で異常検知を行うことによるオーバヘッドを測定した．

Optimization of Intrusion Detection System Based on Static Analyses

Hirotake Abe,[†1] Yoshihiro Oyama,[†2] Mizuki Oka[†3]
and Kazuhiko Kato[†4,†5]

Attacks on programs by exploiting its vulnerability become well-known and severe problem today. These attacks manipulate behavior of vulnerable programs. We have implemented an intrusion detection system that detects the anomaly behavior of programs. The system ensures that the behavior of a program never stray from the control flow extracted from its binary code. Our system utilizes call stack information to detect anomaly behavior. Call stack information is useful to determine current execution state, and therefore our system can achieve higher precision and lower overhead as contrasted with other intrusion detection systems. Furthermore, our system utilizes search history of transition paths of control flow. This technique reduces the overhead further. We show the results of measurement of the overhead.

1. はじめに

バッファオーバフロー攻撃等のプログラムの脆弱性を悪用する計算機システムへの攻撃[27]や，未熟なプログラマが記述した品質が低いプログラムの異常動作による被害が後を絶たない．脆弱性を含むかもしれないプログラムの異常を早期に検知し対策を講じるには，侵入検知システムの利用が有効である．侵入検知システムはプログラムの実行を監視し，異常が発生したと判断されるときにはユーザに警告したりプログラムを終了させたりする等の処理を行う．侵入検知システムが有用であるためには，検知のオーバヘッドが小さいことと，検知精度が高い（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ヘッドが小さく検知精度が高い侵入検知システムの構築である．本研究の対象は侵入検知システムの中の特に異常検知システムと呼ばれるシステムである．異常検知システムはプログラムの正常な動作を表すデータ（そのプログラムの**モデル**）を作り，実行がそのモデルに従っているかどうかを検査することによって異常を検知する．異常検知システ

†1 筑波大学大学院博士課程工学研究科
Doctoral Program in Engineering, University of Tsukuba
†2 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
†3 筑波大学大学院修士課程理工学研究科
Master's Program in Science and Engineering, University of Tsukuba
†4 筑波大学電子・情報工学系
Institute of Information Sciences and Electronics, University of Tsukuba
†5 科学技術振興事業団
Japan Science and Technology Corporation

ムは未知の攻撃・異常を検知できるという特長を持つ．我々が構築する侵入検知システム（異常検知システム）は，プログラムがたどっている制御フローが，バイナリコードが規定する制御フローの規則に従っているかどうかを検査する．攻撃によって乗っ取られたプログラムは多くの場合バイナリコードに反する制御フローをたどるので，制御フローを検査すれば多くの攻撃を検知できる．我々のシステムはシステムコールが呼び出されるごとにその検査を行う．

我々のシステムが防止しようとしている主な攻撃はバッファオーバフロー攻撃である．バッファオーバフロー攻撃は，確保された領域を超える長さのデータをバッファに書き込むことにより，プログラムの内部情報を不正に更新する攻撃である．スタックに保存された返り番地や関数ポインタを更新すれば，プログラムの制御を望みの場所に移動させることができ，攻撃者が望む処理をそのプログラムに実行させること（たとえばシェルを実行させること）ができる．多くのバッファオーバフロー攻撃はバッファに攻撃コードを書き込んだ後にスタック内に保存された返り番地をそのコードの番地に書き換える手段を用いている．バッファオーバフロー攻撃は世界の計算機システムに最も多くの被害を与えている攻撃の 1 つである．バッファオーバフロー攻撃からの防御手段は今までにたくさん提案されてきた．手段の 1 つは侵入検知システムによって攻撃の発生を早期に検知するアプローチであり，本研究ではそれを扱う．他のアプローチのいくつかは 6 章で述べる．

我々のシステムでは Wagner らの方法[23] を改良した方法を用いる．詳細は 2.2 節で述べるが，彼らの方法はシステムコールの情報のみに基づいて，プログラムの制御の場所を非決定的に把握する．その非決定性はオーバヘッドの増加と検知精度の低下をもたらす．一方，我々のシステムは実行時のスタックの情報を利用して制御の場所を決定的に把握する．それはオーバヘッドの削減と検知精度の向上をもたらす．さらに，我々のシステムでは，過去の検査時に判明した部分的な結果を蓄積し，永続化する機能を保有している．その結果は，以降の検査のオーバヘッドを削減するために利用される．

本研究の貢献を以下に示す．

- バイナリコードから抽出した正常な動作に関する情報と実行時のスタックの情報を組み合わせて異常を検査する侵入検知システムを提案する．
- 部分的な検査結果を集積し，再利用することで，検査時のオーバヘッドを削減する．

本論文の構成を以下に示す．2 章で既存の侵入検知システムを説明する．3 章で我々のシステムについて述べる．4 章で実験結果を示す．5 章で議論する．6 章で本研究と関連研究を比較する．7 章でまとめと今後の課題を述べる．

2. 既存の侵入検知システム

2.1 静的解析を用いない侵入検知システム

侵入検知のアイデアは最も早くには 1980 年の文献 2) に出現する．その後文献 5) で，ユーザの動作をそのユーザの正しいとされる動作モデルと照合する等の，侵入検知システムの基礎となる概念が提示された．そして現在も侵入検知システムは活発に研究され進歩し続けている．

いくつかの侵入検知システムは，異常動作を表す規則や異常を特徴づけるデータをプログラムの実行と照合して異常を検知する．大半のアンチウィルスソフトウェアはこの方法を用いている．この方法の欠点は未知の攻撃・異常を検知できないことである．

別の侵入検知システムでは，プログラム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作り，その規則に従って異常を検査する[3),8),11),16)~19),22),25)]．この方法には，正常な動作を学習させるためにプログラムを様々な入力で実行する必要があるという問題と，false positive が出る（正常な実行を異常と判断する）という問題がある．

文献 14)，15)，21) では学習によらない方式が提案されている．この方法では，プログラムの正常な動作の仕様をユーザに記述させ，侵入検知システムはプログラムの動作がその仕様に従っているかどうかを検査する．この方法には仕様を記述する負担がユーザにかかるという問題と，仕様に人間の誤りが入りうるという問題がある．

2.2 静的解析に基づく侵入検知システム

2.2.1 方　　法

近年 Wagner らが静的解析に基づく侵入検知システムを提案した[23]．彼らが用いた方法では，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，ユーザは規則を作る手間を払う必要がない．加えて，彼らの方法には，未知の攻撃を検知できたり，false positive が出ない等の優れた性質がある．彼らの方法は利便性の高いシステムを構築するのに有望であると我々は考える．

彼らの方法では，C プログラムの各関数ごとに，関数呼び出しや条件分岐等の制御フローを表現するプッシュダウンオートマトンが作られる．さらに，プッシュ

```
f(int x) {
  x ? getuid() : geteuid();
}
g() {
  fd = open(...);
  f(0);
  close(fd);
  f(1);
}
```

```
Entry(f) ::= getuid | geteuid
Entry(g) ::= open Entry(f) close Entry(f)
```

図 1 文献 23) の方法で作られる文脈自由文法の例
Fig. 1 An example of context-free grammar made by the method described in Ref. 23).

ダウンオートマトンから，プログラムが呼び出しうるシステムコール列のパターンを表現する文脈自由文法が作られる．文脈自由文法がプログラムのモデルとなる．

彼らのシステムは，コンパイラ部および実行時システムで構成される．コンパイラ部はソースコードを解析して実行可能コードとモデルを生成する．実行時システムはシステムコールが発行されるたびにプログラムを停止させ，システムコール列がモデルに従っているかどうかを検査する．実行時システムは，与えられた文脈自由文法に従ってシステムコール列をパーズするオンラインパーザとして実装されており，パーズできないシステムコール列を異常と判断する．関数定義とその関数定義に対応する文脈自由文法の例を**図 1** に示す．その文脈自由文法をモデルとして異常検知を行う場合，

```
open getuid close getuid
```

というシステムコール列の発行は正常と見なされ，

```
open close getuid getuid
```

というシステムコール列の発行は異常と見なされる．システムコール以外の情報（ヒープやスタックの情報等）は，正常か異常かの判断に利用されない．

2.2.2 問　　題

彼らの方法には 2 つの問題が存在する．第 1 の問題は，実行時オーバヘッドがきわめて大きいことである．たとえば，文献 23) によると，異常検知をしながらの sendmail の実行でメール配送の 1 transaction の処理に 1 時間以上かかっている．これほどオーバヘッドが大きいと，このシステムを実用の場で利用することは非現実的である．

大きいオーバヘッドの原因の 1 つは，プログラムの制御の状態をシステムコールだけから推定しているために，把握される制御の状態が非決定的になり，考慮すべき状態の数が増加していることである．実行時システムは，プログラムを「システムコール列を生成するブラックボックス」として扱い，システムコール列をパーザがパーズできるかどうかだけを検査する．現在の制御の場所は，制御が存在する可能性がある候補のうちのどこかにあるという形で雑に把握する．そして，検査対象のシステムコールを呼び出しうる制御の場所の少なくともどれか 1 つに現在の制御が存在する可能性があれば，実行を正常と見なす．制御が存在しうる全候補を考慮して検査を行うので計算量はきわめて大きい．

```
main() {
  if (...) {
    a();
  } else if (...) {
    b();
  } else {
    c();
  }
}
```

```
a() {
  open();
  read();
}
b() {
  open();
  connect();
}
c() {
  open();
  exec();
}
```

```
Entry(a) ::= open read
Entry(b) ::= open connect
Entry(c) ::= open exec
Entry(main) ::= Entry(a)
              | Entry(b)
              | Entry(c)
```

図 2 文献 23) の方法で制御の状態の把握が非決定的になるプログラムと，そのプログラムのモデルを表す文脈自由文法の例
Fig. 2 An example of programs whose control states are undeterministic while implementing the method described in Ref. 23) and its corresponding context-free grammar.

図 2 のプログラムを例にとって具体的に説明する．そのプログラムでは main 関数内の条件分岐によって関数 a，関数 b，関数 c のいずれかに制御を移す．いずれの場合もまず open を呼び出し，次に read，connect，exec のいずれかを呼び出す☆．彼らのシステムでは，open が呼び出されたとき，制御が a にあるのか b にあるのか c にあるのかを調べない．それらすべての可能性を考慮し，現在の制御の状態に非決定性を残したままで検査を続ける．文脈自由文法で説明するならば，彼らのパーザは，終端記号 open を読んだ後，そ

☆ 現実には，C プログラムにおける open 関数の呼び出しが OS レベルで open システムコールだけを呼び出すとは限らない．しかし，本論文の説明では簡単のために，C プログラムにおける open，read，connect，exec 等の関数の呼び出しは，OS レベルで open，read，connect，exec 等のシステムコールだけを呼び出すと仮定する．

の終端記号 open を生成した非終端記号が Entry(a), Entry(b), Entry(c) のいずれなのかを決定しないまま，次の終端記号の読み込みに移る．大きなプログラムでは，複数の制御の候補から遷移する制御の候補が複数ありえ，さらにその先の遷移先も複数ありえ，考慮すべき遷移が非常に多くなる．上のプログラムでは，open の実行後に制御が存在しうる候補がすべて違うシステムコールを呼び出すので，次のシステムコールが呼び出された時点で，制御の場所は 1 つに確定する．しかし，現実のプログラムではこのようなケースは少ない．

第 2 の問題は検出精度である．図 2 のプログラムで，関数 c の open の実行後に exec が呼び出されるのは正常である．一方，関数 a および関数 b の open の実行後に（バッファオーバフロー等の原因で）exec が呼び出されたら異常と判断すべきである．しかし，彼らの方法は，open の実行後に exec が呼び出されたら，(本当は制御が関数 a や関数 b にあっても）制御は関数 c にあると判断し，実行を正常と見なす．すなわち，彼らの方法は，プログラムがたどりうる全制御フローの中に，検査対象のシステムコール列を生成するものが 1 つでも存在するならば，実行を正常と見なす．文脈自由文法で説明するならば，モデルの文脈自由文法がシステムコール列をパーズできるならば，プログラムの内部でどんなに異常なことが起きていても，実行を正常と見なす．

彼らはこれらの問題に対する 1 つの解決法を提示している．それは，ソースコードの文面上定数であるシステムコール引数と実行時のシステムコール引数を比較して，異なる場所からの同じシステムコールの呼び出しを区別するものである．その解決法は非決定性を減少させる．しかし，すべての非決定性をつねに解消できるわけではない．ソースコードの文面上で定数の引数を与えられていないシステムコールの呼び出しで生じる非決定性は解消できない．

3. 提案方法

3.1 基本的枠組み

我々の方法は監視されているプログラムのスタックの情報を利用して制御の場所を決定的な形で把握することにより，前節に述べた問題を解決する．図 2 の例を用いて説明すると，提案方法は，open を呼び出したのは関数 a，関数 b，関数 c のいずれなのかをスタックを見て調べる．スタックを読む処理それ自体はオーバヘッドを増加させる．しかし，非決定性が解消され考えるべき遷移の数が減れば，全体として性能は

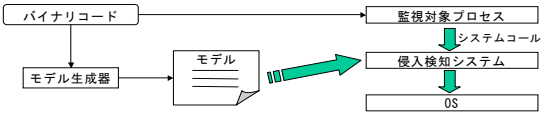

図 3 我々の侵入検知システムの構成
Fig. 3 Composition of our intrusion detection system.

```
copy_file() {
  if (...)
    copy_init();
  do_copy();
  if (...) {
    if (...) {
      error();
    } else {
      rm();
    }
  } else {
    while (...) {
      check();
    }
  }
```

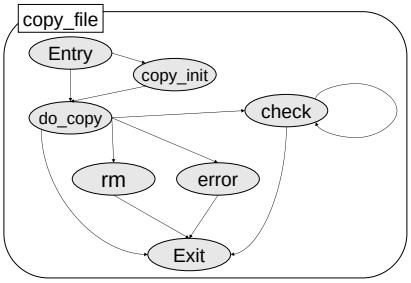

図 4 我々のシステムが作るオートマトンの例
Fig. 4 An example of automata that our system generates.

上がる可能性があると考える．

我々のシステムの構成を**図 3** に示す．我々のシステムは，Wagner らのシステムとは異なり，バイナリコードからモデルを生成する．そのため，ソースコードが入手不可能なプログラムに対しても検査を行うことが可能である．また，ソースコードを用いる方法では対処できない，インライニング等の最適化が行われた場合でも対応できるという利点がある．

モデル生成部は，逆アセンブラを利用して対象のバイナリコードのアセンブリ表現を得る．モデル生成部は，その出力を処理することで制御フローを表すオートマトン（モデル）を作る．関数定義とその関数定義から作られるオートマトンの例を**図 4** に示す．オートマトンの端点は (1) 関数入口，(2) 関数出口，(3) 関数呼び出しのどれかである．関数ポインタによる関数呼び出しは，アドレスを取られたすべての関数を呼び出

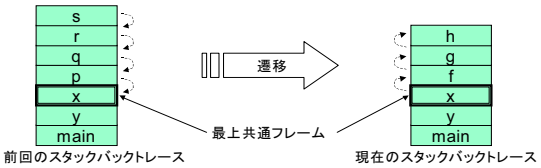

図 5 連続するシステムコール呼び出し時のスタック．我々の方法は点線をたどる制御フローをモデルから探す
Fig. 5 Extracted stacks of two sequential system calls. Our method traces the control flows shown by dotted lines.

す可能性があると見なす．

実行時システムはプログラムがシステムコールを呼び出すたびにプログラムを停止させる．停止させたら，プログラムのスタックの各フレームに保存された返り番地を読むことによって，各フレームが呼び出している関数を調べる．そして，呼び出されている関数のリスト（**スタックバックトレース**）を作る．次に，実行時システムは前回のシステムコール呼び出し時のスタックバックトレースと現在のスタックバックトレースを検査する．前者から後者に至る制御フローがモデルに存在すれば実行は正常と判断し，存在しなければ異常と判断する．

スタックバックトレース間の制御フローがモデルに存在するかどうかは次のアルゴリズムで調べる．

- モデル生成時に，バイナリコード内の各関数を，(1) システムコールを必ず呼び出す関数と，(2) システムコールを呼び出さないかもしれない関数に分類しておく．
- 前回と現在のスタックバックトレースを比較し，底のフレームからどのフレームまでが同じ制御の場所で，どのフレームから制御の場所が異なるかを調べる．同じ制御の場所を持つフレーム群のうち最も上にあるものを**最上共通フレーム**と呼ぶ（**図 5**）．
- 前回のスタックバックトレースからフレーム群をポップして最上共通フレームに至る制御フローを探す．ポップされる各フレームの関数のオートマトンにおいて，フレームに保存された制御の端点から関数出口の端点に至るパスを探す（**図 6**）．パスはシステムコールを呼び出さないかもしれない関数の呼び出しの端点は含んでもよいが，システムコールを必ず呼び出す関数の呼び出しを含んでいてはならない．
- 最上共通フレームの関数のオートマトンにおいて，フレームに保存された制御の端点から，現在のスタックバックトレースにおける最上共通フレームの 1 つ上のフレームをプッシュする関数呼び出し端点に至るパスを探す（**図 7**）．そのようなパスが存在しない場合には，(1) 関数出口に至り，(2) 1 つ下のフレームに戻ってから，再びその関数を呼び出し，(3) 関数入口から望みの関数を呼び出す端点に至るようなパスを探す．
- 最上共通フレームに順にフレームをプッシュして現在のスタックバックトレースを作る制御フローを探す．プッシュされるべき各フレームの関数のオートマトンにおいて，関数入口の端点から，次にプッシュされるべきフレームの関数の呼び出しの端点に至るパスを探す（**図 8**）．パスは，システムコールを呼び出さないかもしれない関数の呼び出しの端点を含んでもよい．

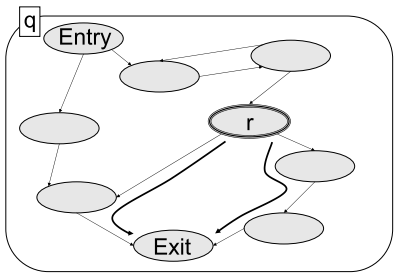

図 6 スタックフレームを pop する制御フローの探索
Fig. 6 Control flow's resulting search path of a popped stack frame.

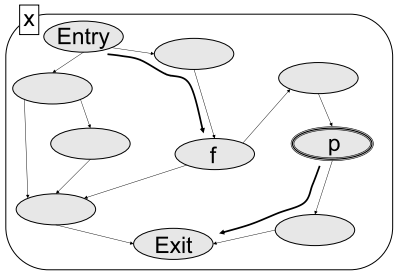

図 7 スタックフレームを一度出て再び入ってから望みの関数を呼び出す制御フローの探索
Fig. 7 Control flow's resulting search path after returning from a called function that has also reached its next calling function.

現在の我々のシステムはユーザレベルスレッドと setjmp/longjmp には対応していない．

3.2 探索履歴の集積と再利用

前述のアルゴリズムによる検査の過程において，以下の部分的な結果が副次的に得られる．

ある関数のオートマトンに含まれている，ある 2 つの端点 a，b について，システムコールをまったく呼ばずに a から b へ遷移する

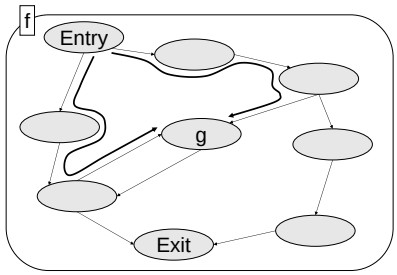

図 8 スタックフレームを push する制御フローの探索
Fig. 8 Control flow's resulting search path of a pushed stack frame.

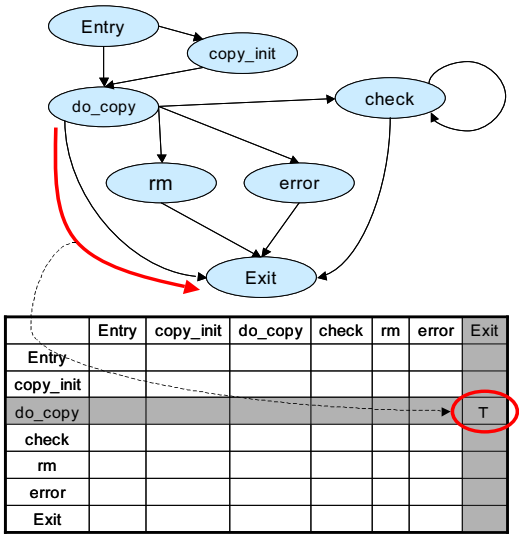

図 9 探索履歴の例
Fig. 9 An example of search histories.

可能性のあるパスが少なくとも 1 つある．

これらの結果を用いると，以前に 1 度行った探索を省略することが可能になり，検査時のオーバヘッドの削減につながる．我々のシステムは，検査中にこの情報（**探索履歴**と呼ぶ）を記憶し，可能な限り探索を省略する．

探索履歴は，モデル内の各関数に 1 対 1 対応する行列の集合として表現される．ある関数のオートマトンに含まれる端点の数を n とすると，その関数に対応する探索履歴は $n \times n$ の正方行列となる．探索履歴内の行列の例を**図 9** に示す．

検査対象プログラムの実行が終了すると，その検査中に得られた探索履歴をファイルに出力され，永続的に保存される．その探索履歴は，対象のバイナリコードが変更されてモデルが変化しない限りつねに有効である．そのため，次回の検査時にはその探索履歴を初めから読み込むことで，いくつかの探索を初回から省略することが可能である．

検査を繰り返し行い，探索履歴が多く蓄積されると，次第に多くの探索が省略されるようになる．その結果，実行時のオーバヘッドは，探索履歴を参照するためにかかる時間だけになり，すべてのパスを探索する場合に比べて大幅に削減されることが期待される．

探索履歴の蓄積は，一種の学習プロセスであるととらえることができる．他の侵入検知システムのいくつか[7),21)]は，学習によってモデルを構築する．それに対し，我々のシステムは，モデル自体は静的解析を用いて生成するが，そのモデルにおける探索結果を学習する．学習によってモデルを構築していくシステムでは，学習が進むほど false positive が減る．我々のシステムでは，学習をまったく行っていない状態でも元々 false positive は出ずに，学習が進むほど検査のオーバヘッドが削減される．

我々は，上記のようなある頂点からある頂点へ遷移する可能性を，プログラムを実行する前にあらかじめ計算しておくという手法も検討した．しかし，そのような計算は，素朴な方法を用いると計算量が膨大になり，計算が終了しない．なぜなら，完全な表を作成するには，関数の入口から出口までの，巡回路を含むものも含めたすべての経路を列挙する必要があるからである．そのため，今回は実際に実行させることでこのような情報を蓄積する方法を採用した．

4. 実　　験

異常検知システムを実装して実験を行った．実験環境は Pentium 4（2.80 GHz）上で動作する Red Hat Linux 9 である．モデル生成器および異常検知システムはそれぞれ Python 言語および C 言語で記述された．モデル生成器は，対象のバイナリファイルを逆アセンブラで処理し，その出力を解析してオートマトンのモデルを生成する．異常検知システムは，linux の ptrace システムコールを利用して監視対象のプログラムの実行を監視する．実装の簡単のため，対象のバイナリファイルは，静的にリンクされたものを使用した．

使用したプログラムは GNU の wc，cp，tar である．wc の実行では合計 3.8 MB のテキストファイルのサイズ・単語数・行数を数えた．cp の実行では約 13 MB のファイルのコピーを行った．tar の実行では約 66 MB のアーカイブファイルを作成した．

表 1 は，元の実行時間と，異常検知しながら走らせたときの実行時間を測定した結果である．測定は，異常検知なし，異常検知ありの場合を計測した．異常検知ありの場合は，探索履歴不使用の場合と，履歴情報

表 1 異常検知により加わる実行時オーバヘッド
Table 1 Runtime overheads incurred by our intrusion detection system.

	wc		cp		tar	
異常検知なし	1.275		3.870		13.16	
異常検知あり：探索履歴不使用	3.491	(2.73)	61.77	(16.0)	85.32	(6.48)
異常検知あり：探索履歴使用，蓄積なし	2.536	(1.98)	29.11	(7.52)	32.83	(2.49)
異常検知あり：探索履歴使用，蓄積あり	1.944	(1.52)	7.560	(1.95)	24.55	(1.87)

時間の単位は秒，括弧内は倍率

使用の場合を計測した．探索履歴使用の場合は，以前に蓄積された探索履歴を使用する場合と使用しない場合を計測した．蓄積された情報は，前回にまったく同じ実行を行った際のものを利用した．

実験の結果から，蓄積された探索履歴情報を使用することで，何も使用しない場合に比較して大幅な性能改善が得られていることが読み取れる．探索履歴がない場合は最小で約 2.7 倍，最大で約 16 倍というオーバヘッドが生じている．プログラムによって差が生じるのは，プログラムの持つ関数の大きさに違いがあることが原因であると考えられる．それは，大きくて複雑な関数ほど，パスの探索範囲が広大になるからである☆．

このような大きなオーバヘッドが，探索履歴を利用することで約 2 倍から約 7.5 倍までに軽減されている．これは，探索履歴を利用することで，2 回目以降はパスの探索が必要なくなるためである．さらに，蓄積された探索履歴を再利用すると，今回は蓄積時の実行と同じ実行なので，パスの探索を行う必要がまったくなくなる．そのため，すべてのプログラムで 2 倍以下までオーバヘッドが軽減されている．

5. 議　　論

検査のタイミング

我々のシステムが異常を検査するのはシステムコール呼び出しのタイミングである．しかし，「動作がバイナリコードから抽出したモデルに従っていることを検査しながらプログラムを実行する」という枠組み自体は，他のタイミングにおける異常の検査を妨げない．たとえば，関数の呼び出しと復帰ごとにプログラムを止めて異常を検査したり，命令実行ごとに異常を検査してもかまわない．しかし，我々はシステムコール呼び出しのタイミングで異常を検査することが最も現実的であり，かつ，十分だと判断した．その理由は次のとおりである．

- システムコールで異常を検査する方法のオーバヘッドは，他の多くの方法（たとえば全部の関数の呼び出しと復帰で異常を検査する方法）のオーバヘッドよりも小さい．
- システムコールでプログラムを停止させる処理は他のタイミングで停止させる処理よりも簡潔に実装できる．
- ほとんどの攻撃はシステムコールの異常をともない，かつ，システムコールを呼び出さずに計算機システムに被害を与えることは難しいので，システムコール呼び出しのタイミングで異常を検査すれば，ほとんどの攻撃を防止できる．

一般に，検査の頻度とオーバヘッドはトレードオフの関係にあるので，状況に応じて適切な頻度を選ぶことが重要である．

正常か異常を判断する材料

システムコールだけしか検査しない侵入検知システム[3),11),16)~19)]でも，単純な攻撃（たとえばバッファをオーバフローさせた後に特に工夫せずに/bin/sh を exec する攻撃）ならば，高確率で検知できる．しかし残念ながら，それらのシステムの検知をすり抜ける工夫を施した攻撃コードを作成することは容易である．文献 24) には，攻撃コードが正常な動作を偽装して，N-gram に基づく侵入検知システム[11)]の検知をすり抜けた例が示されている．我々のシステムはスタックも検査するので，攻撃コードが正常な動作を偽装することが難しい．今後攻撃コードは高度化することが予想されるため，我々の方法や文献 14)，15)，21)，22)，25) の方法のような多様な実行時情報を検査する方法の重要性はますます高まると思われる．ただし，一般に，検知精度とオーバヘッドはトレードオフの関係にあるので，状況に応じて適切な精度を選ぶことが重要である．

サンドボックスとの関係

バッファオーバフロー攻撃への対策としては，サンドボックスを作るシステムも広く用いられている．プログラムをサンドボックス内で実行すれば，バッファオーバフロー攻撃による影響が及ぶ範囲をサンドボッ

☆ プログラム cp のチェックが遅くなっているのは，copy_internal という関数が原因である．関数 copy_internal は，C 言語のソースコードで約 600 行ある．

クス内に限ることができる．サンドボックスを作るシステムの例には，Janus[10]，SoftwarePot[12),26)]，阿部らのシステム[29)]，品川らのシステム[28)] 等がある．また，多くの OS が提供する機能である chroot は，プロセスがアクセスできるファイル名前空間を部分木に限定するものであり，主に ftpd 等のサーバをサンドボックスに閉じ込める手段として利用されている．

サンドボックスと侵入検知システムには利害得失があり，片方があれば他方は不要ということにはならない．サンドボックスはプログラムがアクセスできる資源を制限するものであり，ユーザは「どのプログラムによるどの資源へのアクセスを許すか」というポリシーを定める必要がある．ポリシーを定める作業は単純ではない．アクセスできる資源が少なすぎるとプログラムは動かず，多すぎるとシステムが危険にさらされる．サンドボックスを効果的に使うには，プログラムを深く理解して作られた適切なポリシーが存在するか，サンドボックスの制限を意識してプログラムが作られている必要がある．一方，侵入検知システムはアクセスできる資源の制限を通じてではなく，侵入と判断されるデータや動作の検知を通じてシステムを守る．侵入検知システムのユーザは，「プログラムを満足に動かすにはどの資源へのアクセスを許す必要があるか？」等の点に悩む必要はない．

危険なシステムコールの実行を防ぐための Janus や SoftwarePot 等のサンドボックスには別の問題もある．それは，資源へのアクセスを許すかどうかを，プログラムの実行状態を考慮せずに判断することである．たとえば，あるファイルへの更新を制御する場合，プログラムの全実行を通じて更新を許すか，全実行を通じて更新を禁止するかのどちらかしか選択できない．これは Netscape 等の複雑な動作を行うプログラムの実行で不都合をもたらす．Netscape を実行するには，サンドボックスは Netscape に /etc/passwd への読み出し許可を与える必要がある[1)]．しかし，許可を与えると，Netscape がバッファオーバフロー攻撃等によって乗っ取られたとき，攻撃者に /etc/passwd を読まれてしまう．本研究のシステムのように，許可される操作が実行状態に応じて動的に変化するシステムでは，そのような事態を避けることができる．

6. 関 連 研 究

本章では 2 章で触れなかった関連研究について述べる．

最近，Feng らによって，我々のシステムと同様にスタック情報を利用するシステムが発表された[7)]．彼らの方法は，学習によってモデルを作成するという点で我々の方法と異なっている．学習では完全なモデルを作成することが事実上不可能なので，どれだけ学習を繰り返しても false positive が発生する可能性をなくすことはできない．それに対し我々の方法では，初めから false positive が出ないということを保証できるという利点がある．

文献 9) の研究では，Wagner らの方法を改良した方法を，遠隔計算機に投入したジョブからローカル計算機への遠隔システムコール呼び出しの異常の検知に適用している．彼らは制御の場所に関する非決定性を減らす最適化として，wrapper 関数やダミーな遠隔呼び出しを挟むソースコード変換等を提案している．本研究の方法と彼らの最適化は共存可能である．

Program shepherding[13)] は，制御移動命令のインタプリティングによって，制御の移動がセキュリティポリシーに従うことを強制する枠組みである．実装上は制御移動命令を毎回インタプリティングするわけではなく，動的コード生成技術を利用してできる限りネイティブコードを実行するようにしている．彼らが扱っているセキュリティポリシーは，ブランチ命令が関数先頭のアドレスに飛ぶことや，関数復帰命令が呼び出し命令の後ろに飛ぶこと等の，個々のプログラムの特徴によらない一般的なものだけである．ソースコードから抽出した制御フローをたどるという本研究の「セキュリティポリシー」は扱われていない．しかし，彼らのシステムに改造を加えて，本研究のポリシーを program shepherding によって強制することは可能だと我々は推測している．

Non-executable stack[20)] は，スタック領域にあるコードを実行できないようにする OS の機構である．この機構は関数ポインタを改変する攻撃や，return-into-libc と呼ばれる攻撃を防げない．return-into-libc 攻撃とは，返り番地を libc の system のようなライブラリ関数の番地に書き換えることによって任意のコマンドを実行させるものである．我々のシステムではそのような攻撃も検知できる．

StackGuard[4)] と propolice[6)] 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．これらが生成するコードはスタック以外にあるデータ（たとえばヒープ上にある関数ポインタ）の改変による攻撃を検知できない．我々のシステムではそのような攻撃も検知できる．

7. まとめと今後の課題

バイナリコードの情報と実行時のスタックの情報を照合することによってプログラムがたどっている制御フローの異常を検知するシステムを提案した．静的解析に基づく侵入検知システムは実社会で広く使われ役立つシステムになりうると考えるが，Wagner らの方法はオーバヘッドが大きすぎて実用的ではない．実用に使える静的解析に基づく侵入検知システムを構築するのが我々の長期的目標であり，本論文ではその目標に近づくための技術の 1 つを示した．我々のシステムの特長を以下にまとめる．

- 学習をしなくても false positive が出ない．
- Wagner らの方法に比べ，異常を検査する処理において考慮する必要がある制御の場所の数が少ない．それはオーバヘッドの削減と検知精度の向上につながる．
- 攻撃コードが正常な実行を偽装することが難しい．我々のシステムはシステムコール列だけでなくスタックも検査するので，異常な実行を正常な実行に偽装するための攻撃者の手間が大きい．
- 学習によって探索履歴を蓄積することで実行時オーバヘッドが削減される．

我々のシステムを用いて実験を行ったところ，異常検知によるアプリケーションの実行時間の増加は 2 倍程度に抑えることが可能であることが分かった．

今後は，より大規模なアプリケーションで実験を行いたい．次に，Wagner らの方法に基づくシステムを実装して実験を行い，彼らの方法と我々の方法の性能を比較したい．また，現在は学習によって蓄積している探索履歴を，あらかじめオートマトンから計算しておくためのアルゴリズムの検討を行いたい．さらに，現在の我々のシステムではシステムコールの引数等のデータの中身の情報を利用していないが，今後それらの情報を活用して検知精度を高めていきたい．

参 考 文 献

1) Acharya, A. and Raje, M.: MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications, *Proc. 9th USENIX Security Symposium*, Denver (Aug. 2000).
2) Anderson, J.P.: Computer security threat monitoring and surveillance, Technical Report, James P. Anderson Company (Apr. 1980).
3) Asaka, M., Onabuta, T., Inoue, T., Okazawa, S. and Goto, S.: A New Intrusion Detection Method Based on Discriminant Analysis, *IEICE Trans. Inf. Syst., Special Issue on High-speed Internet Technology and its Applications*, Vol.E84-D, No.5, pp.570–577 (May 2001).
4) 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 Symposium*, San Antonio, pp.63–78 (Jan. 1998).
5) Denning, D.E.: An Intrusion-Detection Model, *IEEE Trans. Softw. Eng.*, Vol.13, No.2, pp.222–232 (1987).
6) Etoh, H.: GCC extension for protecting applications from stack-smashing attacks. http://www.trl.ibm.co.jp/projects/security/ssp
7) Feng, H., Kolesnikov, O., Fogla, P., Lee, W. and Gong, W.: Anomaly Detection using Call Stack Information, *Proc. IEEE Security and Privacy 2003*, Oakland (May 2003).
8) Ghosh, A.K., Schwartzbard, A. and Schatz, M.: Learning Program Behavior Profiles for Intrusion Detection, *Proc. Workshop on Intrusion Detection and Network Monitoring*, Santa Clara, pp.51–62 (Apr. 1999).
9) Giffin, J.T., Jha, S. and Miller, B.P.: Detecting Manipulated Remote Call Streams, *Proc. 11th USENIX Security Symposium*, San Francisco (Aug. 2002).
10) 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 Symposium*, San Jose, pp.1–13 (July 1996).
11) 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).
12) 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 (Feb. 2003).
13) Kiriansky, V., Bruening, D. and Amarasinghe, S.: Secure Execution Via Program Shepherding, *Proc. 11th USENIX Security Symposium*, San Francisco (Aug. 2002).
14) Ko, C., Fink, G. and Levitt, K.: Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring, *Proc. 10th Annual Computer Security Applications Conference*, Orlando, pp.134–144 (Dec. 1994).
15) Ko, C., Ruschitzka, M. and Levitt, K.: Execution Monitoring of Security Critical Programs in Distributed Systems: A Specification-based

Approach, *Proc. 1997 IEEE Symposium on Security and Privacy*, pp.175–187 (May 1997).
16) Kosoresow, A.P. and Hofmeyr, S.A.: Intrusion Detection via System Call Traces, *IEEE Software*, Vol.14, No.5, pp.35–42 (1997).
17) Lee, W. and Stolfo, S.J.: Data Mining Approaches for Intrusion Detection, *Proc. 7th USENIX Security Symposium*, San Antonio (Jan. 1998).
18) Liao, Y. and Vemuri, V.R.: Using Text Categorization Techniques for Intrusion Detection, *Proc. 11th USENIX Security Symposium*, San Francisco (Aug. 2002).
19) Marceau, C.: Characterizing the Behavior of a Program Using Multiple-Length N-grams, *Proc. 2000 Workshop on New Security Paradigms*, Cork, Ireland, pp.101–110 (Sep. 2000).
20) The Openwall Project: Linux kernel patch from the openwall project. http://www.openwall.com/linux/
21) Sekar, R., Bowen, T. and Segal, M.: On Preventing Intrusions by Process Behavior Monitoring, *Proc. 1st Workshop on Intrusion Detection and Network Monitoring*, Santa Clara (Apr. 1999).
22) Sekar, R., Bendre, M. and Bollineni, P.: A Fast Automaton-Based Method for Detecting Anomolous Program Behaviors, *Proc. 2001 IEEE Symposium on Security and Privacy*, Oakland, pp.144–155 (May 2001).
23) Wagner, D. and Dean, D.: Intrusion Detection via Static Analysis, *Proc. 2001 IEEE Symposium on Security and Privacy*, Oakland, pp.156–168 (May 2001).
24) Wagner, D. and Soto, P.: Mimicry Attacks on Host-Based Intrusion Detection Systems, *Proc. 9th ACM Conference on Computer and Communications Security*, Washington DC (Nov. 2002).
25) 大山恵弘，王　維，加藤和彦：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，コンピュータシステム・シンポジウム論文集，横浜，pp.45–52 (Nov. 2002).
26) 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).
27) 脇田　建：バッファ溢れ攻撃とその防御，コンピュータソフトウェア，Vol.19, No.1, pp.49–63 (2002).
28) 品川高廣，河野健二，益田隆司：実行可能コンテンツの安全な実行環境，情報処理学会論文誌，Vol.43, No.6, pp.1677–1689 (2002).
29) 阿部洋丈，加藤和彦：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，コンピュータソフトウェア，Vol.20, No.3 (2003).

(平成 15 年 8 月 2 日受付)
(平成 15 年 9 月 10 日採録)

阿部　洋丈

1976 年生．1999 年筑波大学第三学群情報学類卒業．同年筑波大学大学院工学研究科博士課程入学．分散システム，コンピュータセキュリティに興味を持つ．

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）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 年より東京大学大学院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．

岡　　瑞起

1980 年生．2003 年筑波大学第三学群情報学類卒業．同年筑波大学大学院理工学研究科修士課程入学．興味はセキュリティ，データマイニング，パターン認識．

加藤　和彦（正会員）

1962 年生．1985 年筑波大学第三学群情報学類卒業，1987 年工学修士（筑波大学大学院工学研究科），1992 年博士（理学）（東京大学大学院理学系研究科）．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．オペレーティングシステム，プログラミング言語システム，データベースシステム，分散システム，モバイルオブジェクト計算に興味を持つ．電子情報通信学会，日本ソフトウェア科学会，ACM，IEEE 各会員．