システムコールの実行順と実行位置に基づく侵入検知システムの実現

槙本裕司[†]　齋藤彰一[†]　松尾啓志[†]

プログラムの脆弱性を利用して，プログラマの意図に反する異常な動作をさせる攻撃による被害が問題になっている。近年はゼロデイ攻撃が目立つようになった．従来，このような攻撃の検知には，攻撃プログラムのパターンとのマッチングが用いられてきた．しかし，マッチングに基づくセキュリティシステムでは，ゼロデイ攻撃を検知することが難しい．このような攻撃を検知することのできるシステムに異常検知型侵入検知システムがある．本論文ではシステムコールの呼び出し順序とスタックの情報を用いた侵入検知システムを提案する．具体的には，システムコールと，その呼び出し時の実行位置情報を組にしたデータを基にした N-gram 集合と，静的解析により得た情報から，異常な動作か否かを判断する．実験では検知に要するオーバヘッドの測定と，N-gram 集合に実行位置情報も用いたことによる効果についての評価を行った．

Realization of Intrusion Detection System Based on Pattern and Situation of System Call

Yuji Makimoto,[†] Shoichi Saito[†] and Hiroshi Matsuo [†]

In recentry, zero-day attack is increasing. It is difficult for security system based on matching of pattern to detect the zero-day attack. An effective countermeasure against zero-day attack is intrusion detection system with anomaly detection. In this paper, we propose an intrusion detection system based on pattern and situation of system call. Through the experiments, we have evaluated the delay from detection and the effective of improved N-gram method.

1. はじめに

プログラムの脆弱性を利用した乗っ取り攻撃などによる被害が問題となっている．近年はゼロデイ攻撃が目立つようになってきた．ゼロデイ攻撃も検出することができるシステムに異常検知 (Anormaly Detection) システムがある．異常検知はプログラムの正常な動作に基づく規則をあらかじめ作成しておき，その規則と異なる動作を異常と判断するシステムである．これは，あるプログラムに対する異常な動作や, その要因となるセキュリティホールをすべて把握することは不可能であるが，正常な動作を定義することは可能なことを利用している．また正常な動作規則は，一度規則を作成したら更新することなく使いつづけることも可能である．

異常検知型侵入検知システムには，監視対象プログラムの正常な実行におけるシステムコールの呼び出し列を収集し，N-gram 法[1] により特徴化するとによって正常な動作を表す規則を作成する手法が存在する．しかし，システムコールの呼び出し順序のみから N-gram 法を用いて作成された正常な動作規則では，攻撃コードが正常な動作を偽装することで，検知システムに検知されないようにすることも可能である[2)3)]．これはシステムコールの呼び出し順序のみからなる N-gram により作成した規則では，システムコールを呼び出したときのプログラムの実行状態 (例えば，システムコールの呼び出し元の関数) を考慮することができないためである．つまり，システムコールの呼び出し列が同じパターンであっても，プログラムの同じ場所から呼び出されているシステムコール列か否かを保証することができないため，それだけでは高い検出精度を得られない．

さらにこのような手法では，未学習の動作が学習済みの動作規則に一致する可能性が高い．しかし，未学習動作は学習済みの動作とは別に扱う必要がある．なぜなら，システムコールの呼出列が同一であったとしても，実行プログラム上の異なる場所から呼び出されたシステムコールを同等に扱うことは，呼び出し列の偽造のしやすさに関わってくる．

そこで本稿では，呼び出されたシステムコールと，そのときの呼び出し元のユーザ定義関数の位置情報 (戻りアドレス) をひとつの組として，その組の呼び出し列を基に N-gram 法により規則を作成する方式を提案する．システムコールが呼び出された位置情報を付加することで，プログラムの実行位置に基づく規則を作成することができ，より正確な実行パターンを表現することができる．そのため false negative の減少が期待できる．

† 名古屋工業大学
Nagoya Institute of Technology

しかし，システムコールとそのときの呼び出し元の位置情報を組とすると，N-gram により生成される要素の種類が多くなるため，規則のサイズが大きくなる．また，未学習の動作が学習済みの動作パターンに一致する確率が減るため，false positive の増加も考えられる．そこで，未学習の動作が発生した場合，予め対象プログラムの静的解析により作成しておいた規則を用いて，その動作の正当性を評価する手法を併せて提案する．

本システムでは次の 3 つの正統性を確認することでプロセスの異常動作を検出する．

(1) システムコールと呼び出し位置情報の実行順序
(2) システムコールの呼び出し元であるユーザ定義関数の実行位置情報
(3) システムコールの呼び出しに至るまでに呼び出されているライブラリ関数

過去の動作から N-gram 法により特徴を抽出して作成したシステムコールの呼び出し順を表す規則と，静的解析により作成したシステムコールを呼び出したときのユーザ定義関数の実行位置の規則，呼び出されているライブラリ関数の情報からなる規則の 3 つを用いて, 呼び出されたシステムコールの正統性を総合的に判断する．

未学習の動作は静的解析により作成された規則により, 異常か未学習のかを判断することができる．これにより false positive を減少させることが可能である．

本論文では 2 章で既存の侵入検知手法について述べる．3 章で我々の提案手法について述べる．4 章で実験結果を示す．5 章でまとめと今後の課題を述べる．

2. 既存の手法

異常検知を用いた侵入検知システムには正常な動作を学習，解析により規則を作成する手法と，実行プログラムの静的解析により規則を作成する手法の 2 種類がある．この章ではそれぞれの特徴と，問題点について述べる．

2.1 プログラム実行の解析と学習に基づいた侵入検知システム

プログラム実行の解析と学習に基づいた侵入検知システムでは，対象となるプログラムの正常な実行時の動作を解析し，正常な実行を特徴づける動作パターンを作成する[1)4)5)6)]．そのために，対象となるプログラムに可能な限り多種多様な入力をあたえて実行し，パターンを作成，収集することで，正常な動作の規則を学習させる．正常な動作の学習は異常な動作の起こらない環境で行う必要がある．この手法では，完全な学習を保証することが非常に困難であるため，学習できなかった正常な動作に対する措置が必要となる．

正常な動作規則の作成アルゴリズムには，N-gram を用いる方式[1)] の他に，有限オートマトンを用いる方式[4)] や可変長の N-gram を用いる方式[5)] などがある．学習により作成された規則を用いた異常検知方式では，false positive が発生する．未学習の動作が存在する可能性は十分にあるので，規則に反した動作であっても直ちに異常と判断することはできない．そのために，いくつかの手法が提案されている．1) なんらかの情報から異常動作であるか未学習の動作であるかを判断する手法，2) 規則に反した動作にペナルティをあたえ，ペナルティが一定値に達したときに異常と判断する手法，3) 規則に含まれないシステムコールの呼び出し列が一定時間内に閾値以上観測されたら異常と判断する手法[6)]，などである．ペナルティの蓄積により，異常動作の判断をする手法の場合では，異常と判断できるまでには，いくつかの怪しいシステムコールの呼び出しがあったということになるので，異常が検知できたとしても，早急な対処ができないという問題がある．

2.2 静的解析を用いた侵入検知システム

静的解析を用いた侵入検知システムでは，対象となるプログラムのソースあるいは実行ファイルの解析により規則を作成する[7)8)9)]．この手法では実行ファイルに基づくすべての動作パターンを考慮することができるため，基本的には規則と比較し，一致しているかどうかを調べるだけでよい．一致していれば正常，異なっていれば異常と一意に決めることができる．したがって，false positive が発生しないという特徴がある．しかし，プログラムの発生頻度の高い動作や，状況を考慮することが難しい．また false negative が起きる可能性がある．

静的解析を用いた侵入検知システムでは，システムコールの呼び出し順のみを確認する方式[7)] や，スタックの情報を用いる方式[8)] などが考えられている．スタックの情報を用い方として，Feng らが提案するシステム[9)] では Virtual Stack List を生成し，プログラムの実行による Virtual Stack List の変化から Virtual Path を作成している．また，阿部らの手法[8)] では，前回システムコールが呼び出されてから，今回システムコールが呼び出されるまでの関数遷移パスを確認することによって，異常な動作を検出している．

3. 提 案 手 法

本稿では学習と静的解析を組み合わせた規則に基づいた侵入監視システムを提案する．学習により作成された規則は，プログラムの実際の動作に基づき，かつプログラマの意図した動作のみを含む規則を作成できる．しかし，学習による規則に基づく侵入検知システムでは，未学習の動作による false positive が起こる可能性がある．

また，システムコールの呼び出し順のみに基づく規則では，異なる実行状況から呼び出されたシステムコールであっても，システムコールの呼び出し順が同じになる可能性がある．そこで本提案では，ユーザ定義関数の実行位置情報を加えることでプログラムの実行状況が特定できると考え，システムコールの呼び出し順と組み合わせることで，プログラムの実行フローを厳密に定義する規則を作成する．本章では，提案方式の概要を述べた後で，詳細な方式と監視方式について述べる．

3.1 概　　要

提案手法では，次の 3 つの正当性を確認することで，

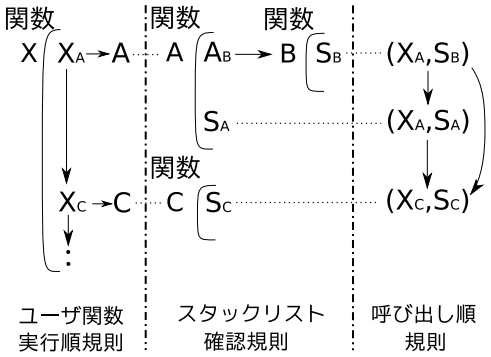


図 1 各規則の関係

プログラムの実行の正当性を保証する．また，それぞれの正当性を確認するために，対応する規則を設ける．この規則名を括弧内に，各規則の関係を図 1 示す．図 1 の関数 X はユーザ定義関数，関数 A,B,C はライブラリ関数，S はシステムコールである．また，X_A は関数 X が関数 A を呼び出すアドレスを，S_A は関数 A から呼び出されたシステムコールを表す．

(1) システムコールの呼び出し順序の確認 (呼び出し順規則)

(2) システムコールの呼び出し元であるユーザ定義関数の実行位置 (アドレス) の確認 (ユーザ関数実行順規則)

(3) システムコールを呼び出したときにスタックに積まれているライブラリ関数の戻りアドレスの確認 (スタックリスト確認規則)

(1) ではシステムコールが呼び出される順番が正しいかどうかを確認する．図 1 の場合の呼び出し順規則は，S_B,S_A,S_C または S_B,S_C の順でシステムコールが呼び出されることを表す．また，呼び出されるシステムコールの番号の他に，呼び出し元のユーザ定義関数の実行アドレスを持つ．図 1 の場合では，S_A 及び S_B は X_A が，S_C は X_C がユーザ定義関数の実行アドレスである．システムコール番号と実行アドレスを基に，その呼び出し順を確認する．

(2) では，システムコールの呼び出し元のユーザ定義関数が正しいか否かを確認する．このために，ユーザスタックを調べ，スタックの最も低位アドレスにあるユーザ定義関数への戻りアドレス (実行位置) を取得する．ユーザ関数実行順規則は，ユーザ定義関数からの全関数の呼び出し順，それらを呼び出している実行アドレスを表している．図 1 のユーザ定義関数 X は，アドレス X_A でライブラリ関数 A を呼び出した後，アドレス X_C でライブラリ関数 C を呼び出すことを表す．

(3) ではユーザ関数実行順規則で確認されたアドレスから呼び出されている最初のライブラリ関数から，システムコールを実際に呼び出した最後のライブラリ関数までの関数の呼び出し順が正いものであるかを，スタックリスト確認規則を用いて確認する．スタックリスト確認規則は，ライブラリ関数が呼び出す関数とそれらの呼び出しアドレスの対応を表している．図 1 のライブラリ関数 A は，次の 2 つの呼び出しが起こる可能性があることを表す．

- アドレス A_B でライブラリ関数 B を呼び出す
- システムコール S_A を呼び出す

また，この規則はスタックリストの確認に用いる．なおスタックリストとは，Freg らのシステム[9] で使用されている Virtual Stack List を簡略化したものである．詳細については 3.4 節で述べる．

各規則とプログラムの挙動を比較した結果，異常と判断されたら呼び出されているたシステムコールの処理が行われる前に，対象のプロセスにたいしてなんらかの対策を施すことができる．

なお，異常動作の検知に，システムコールの呼び出し順を用いることの有効性は，Forrest[1] らの研究によって確認されている．また，システムコールとスタック上に積まれた戻りアドレスは強い相関があることも知られている[9]．本システムでは，呼び出し順規則によるシステムコールの呼び出し順の確認と，ユーザ関数実行順規則及びスタックリスト確認規則による関数の呼び出しを確認することで，高い検知精度が得られる．

3.2 規則の生成手法

本システムで提案する 3 つの規則について，その生成手法を述べる．

3.2.1 呼び出し順規則

呼び出し順規則の作成は N-gram 法を用いる．N-gram を用いた侵入検知システムでは，システムコールの呼び出し列に含まれる長さ N の連続する呼び出し列をすべて捜し出し，実行時にその列と実際の呼び出しを比較する．本システムでは，N-gram で扱う情報に，システムコールの番号とユーザ定義関数の実行位置の組を用いる．例えば，呼び出されたシステムコールの番号が 5，スタック解析により得られたユーザ定義関数の実行位置 (アドレス) が 0x804AAAA であった場合，N-gram で扱う情報の組を以下のように表す．

```
(実行位置，システムコール番号)
= (0x804AAAA, 5)
```

本システムでは次の要素 1，2 のような場合，システムコール番号の並びだけをみると同一であるが，実行位置情報が異なっているため別の要素として扱う．これにより，要素の数は増加するが，より厳密な規則となる．

```
要素 1:
(0x804AAAA,5) (0x804BBBB,3) (0x804CCCC,4)
要素 2:
(0x804AAAA,5) (0x804DDDD,3) (0x804EEEE,4)
```

3.2.2 ユーザ関数実行順規則とスタックリスト確認規則

ユーザ関数実行順規則とスタックリスト確認規則の作成は，簡単のため，静的にリンクされた実行ファイルのディスアセンブル結果に対して静的解析を行い，規則を作成する．ユーザ関数実行順規則は，実行ファイルの jump 命令に基づいて，関数の呼び出し順の関係を調べることで規則を作成する．なお規則の作成範囲は，各ユーザ定義関数の始まりから終了までである．この規則により，ユーザ定義関数からの関数の呼び出し順，及びその関数を呼び出すアドレスを調べることができる．

スタックリスト確認規則は，ライブラリ関数ごとに call 命令で呼び出す関数のアドレスと，その命令の実行アドレスを調べ，規則を作成する．この規則により，あるライブラリ関数から呼び出される関数と，その呼び出しがあったときにスタックにつまれる戻りアドレスを調べることができる．

3.3 規則のデータ構造

3.3.1 呼び出し規則

呼び出し順規則は，N-gram 法により作成した部分列 N の要素の一覧である．N-gram 法に用いるデータは，システムコール番号と，ユーザ定義関数の実行位置情報の組の呼び出し列である．

3.3.2 ユーザ関数実行規則

ユーザ関数実行順規則は、関数の呼び出し毎に 1 つの規則を生成する．これを規則ブロックと呼ぶ．各規則ブロックにはシステムコール番号とスタック情報，次の規則ブロックへの相対アドレスを記録する．規則はすべての関数呼び出しに対応する規則ブロックによって構成される．

本規則を用いることで，前回システムコール呼び出時に一致した規則ブロックから，今回のシステムコール呼び出しに一致する規則ブロックまでの遷移を確認することができる．ある規則ブロックから遷移可能な規則ブロックは，規則ブロック全体の数に比べてかなり限定されるので，その確認は短時間で可能である．

3.3.3 スタックリスト確認規則

スタックリスト確認規則は各ライブラリ関数ごとに作成する．各ライブラリ関数についての規則には以下の情報が含まれる．

- ライブラリ関数の先頭アドレス
- call 命令により呼び出される関数のアドレスとその命令を実行するアドレス
- 呼び出されるシステムコール番号

3.4 監 視 方 法

本節では，監視対象のプログラムの動作について，各規則との比較を行うためのデータ収集と，収集した情報と各規則の比較をするアルゴリズムについて述べる．

3.4.1 プログラムの動作解析によるデータ収集

各規則との比較を行うには，1) プログラムがシステムコールを呼び出す直前のレジスタの値を取得する．2) そのレジスタの値をもとにユーザスタックを探索し，ユーザ定義関数までの戻りアドレスリスト (スタックリスト) を作成する．以上 2 つの処理が必要である．1) について，本システムでのレジスタの値の取得法の詳細は 4.1 節で述べる．2) では，ebp レジスタの値からスタックを探索することで，次の関数フレームの戻りアドレスの位置を知ることができる．そのため，少ないスタック参照でスタックリストを作成することができる．図 2 はスタックとスタックリストの構造である．図 2 の Call Stack は，次の動作後のスタックの状態である．

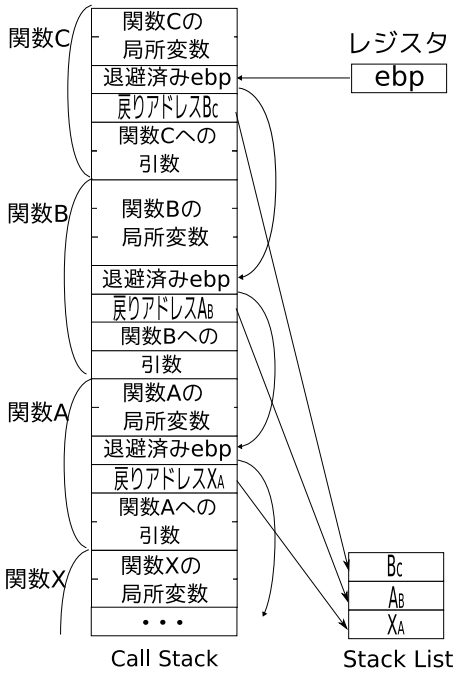


図 2 Stack List の作成

(1) ユーザ定義関数 X が関数 A を呼び出した
(2) 関数 A が関数 B の呼び出した
(3) 関数 B が関数 C の呼び出した
(4) 関数 C がシステムコールの呼び出した

ここで作成されるスタックリストは X_A,A_B,B_C となる．

なお，ユーザ定義関数内のアドレスか否かの判断は，静的解析結果に基づいている．つまり，対象プログラムの実行ファイルを静的解析したときに，ユーザ定義関数のアドレスの範囲をあらかじめ調べておき，それと比較することでユーザ定義関数への戻りアドレスか否かを判断する．

3.4.2 各規則との比較

呼び出されたシステムコールが正常な動作によるものか否かは次のアルゴリズムで確認する．

(1) 過去 N 個 (N-gram で扱う部分列の長さが N の場合) のシステムコール番号とその呼び出し時のユーザ定義関数への戻りアドレスの組の呼出列を作成する．
(2) (1) で作成した呼び出し順が，呼び出し順規則に一致するかを確認する．
(3) (2) で正しいことが確認されたら，システムコー

ルが呼び出されたときのユーザ定義関数の実行位置を，ユーザ関数実行順規則と比較して確認する．

(4) (3) で正しいことが確認されたら，今回のシステムコールが呼び出された時のユーザ定義関数の実行位置から，システムコールが呼び出されるまでのライブラリ関数の呼び出し順をスタックリスト確認規則と比較し，確認する．

(5) (4) で正しいことが確認されたら，正常な動作であると判断する．

確認アルゴリズムの (2) で呼び出し列が一致しなければ，未学習もしくは異常動作である．これらの確認アルゴリズムは，まず，今回システムコールを呼び出したときのユーザ定義関数上への戻りアドレスがユーザ関数実行順規則に含まれるかを確認する．含まれる場合は，未学習だが，正規のユーザ定義関数から正規のライブラリ関数を経由してシステムコールが呼び出されていることを意味する．そのため，前回システムコールが呼び出されたとき (S_{n-1}) のユーザ定義関数の実行位置 (X_{n-1}) から，今回システムコールが呼び出されたとき (S_n) のユーザ定義関数の実行位置 (X_n) までの関数遷移を確認し，その制御フロー上の距離を算出する．S_{n-1} から S_n までの関数遷移の確認は，S_{n-1} のときの実行位置 X_{n-1} から，S_n のときの実行位置 X_n への遷移を調べる．$X_{n-1} = X_n$ であれば，再び X_{n-1} からシステムコールを呼び出したライブラリ関数までのライブラリ関数の制御フローを調べる．また，$X_{n-1} \neq X_n$ であれば X_{n-1} から X_n の制御フローと，X_n からシステムコールを呼び出した関数までの関数の呼び出しを調べる．X_{n-1} から X_n までの制御フロー上の距離が短いほど，正常な動作である確率が高い．X_n が見付からない (ユーザ関数実行順規則に X_n が含まれない) か，X_{n-1} から X_n への制御フローがユーザ関数実行順規則からは確認できない場合は異常な動作と判断する．

また，確認アルゴリズム (4) でシステムコールが呼び出されるまでのライブラリ関数の実行フローが存在しなければ異常なライブラリ関数の呼び出しがあったと判断する．

3.5 本システムの適用範囲

提案した侵入検知システムでは，バッファオーバーフローにより戻りアドレスを書き換え，スタックやヒープ領域にからコードを実行するような攻撃を高確率で検知することができる．また，バッファオーバーフローにより，戻りアドレスを任意のライブラリ関数のアドレスに書き換えるような攻撃でも検知できる確率は高いと考えられる．しかし，関数を呼び出すときの引数を書き換えて，動作を変えるような攻撃は多くの場合，防ぐことはできない．

4. 実験と考察

4.1 実験概要

本節では，3 章で述べた提案手法を Linux 上に実装して行った実験について述べる．なお，今回は侵入検知プログラムを ptrace を用いてユーザプロセスとして実装した．

実験環境は Pentium4(3.40GHz) 上で動作している Fedora Core 5 である．監視対象としたのは GNU の wc である．wc はライブラリを静的にリンクしてコンパイルしたものを解析対象とした．また，N-gram の N=3 として，規則の作成を行った．

4.1.1 実験手順

実験手順を以下に示す．

(1) wc の実行ファイルの静的解析により，ユーザ関数実行順規則と，スタックリスト確認規則を作成する．

(2) 正常な wc の動作で呼び出されるシステムコール，及び実行位置情報を収集し，呼び出し順規則を作成する．なお，学習させる実行パターンを変えて，異なる 2 種類の呼び出し順規則を作成する．

(3) 作成した規則を用いて，侵入検知プログラムで wc を監視しながら実行し，その実行時間を測定する．

(2) で作成する規則は，以下の 2 種類の動作 (動作 1，動作 2) について学習により作成した．いずれの動作も同一の約 15MB のテキストファイルを処理対象とした．

動作 1: オプションなしで `wc` を実行
動作 2: `-l` オプションをつけて `wc` を実行

動作 1 および動作 2 から作成される規則は異なっており，動作 1 から作成される N-gram 集合 A と，動作 2 から作成される N-gram 集合 B の関係は，$A \not\subseteq B$ かつ $A \not\supseteq B$ である．それゆえ，動作 1 を学習対象とした規則を用いて動作 2 の検知を行ったとき，規則とは一致しない動作パターン (システムコールの呼び出し列) が現れる．また，逆も同様である．

4.1.2 侵入検知プログラムの動作

侵入検知プログラムで ptrace を用いて，監視対象プログラムがシステムコールを呼び出したときの各種情報を調べる手順を図 3 に示す．監視対象プログラムはシステムコールを呼び出すと，そのシステムコールのカーネルでの処理の直前で停止し，監視プログラムにシグナルが送信される．監視プログラムは停止している監視対象プログラムのレジスタ，スタックの情報を調べ，スタックリストを作成し，規則との比較を行う．

4.2 侵入検知プログラムのオーバヘッドの測定

測定結果を表 1 に示す．表 1 の「未学習の動作パターンなし」は，規則作成に用いた動作と，監視対象プログラムの動作が同じ場合である．そのため，監視対象プログラムを監視しても，未学習の動作パターンが現れることはい．また，「未学習の動作パターンあり」は，規則作成に用いた動作と，監視対象プログラムの動作が異なる場合である．監視対象プログラムの監視では，未学習の動作パターンが複数回発生した．

動作 1 の場合，呼び出されたシステムコールの数は

表 1　監視による実行時間の増加率

	動作 1	動作 2
侵入検知なし	1.510(1.00)	0.041(1.00)
侵入検知あり: 未学習の動作パターンなし	1.586(1.05)	0.157(3.83)
侵入検知あり: 未学習の動作パターンあり	1.631(1.08)	0.177(4.32)

単位は秒，括弧内は倍率

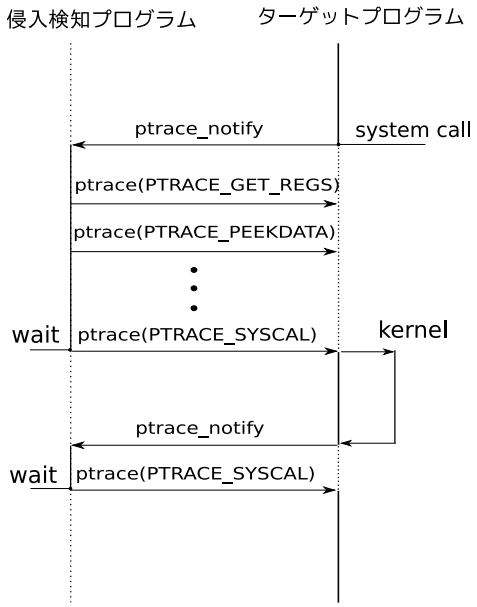


図 3　ptrace による監視手法

974 個，未学習の動作パターンありの場合ではそのうちの 9 個の呼び出しで，システムコール呼び出し順規則に一致しなかった．動作 2 の場合では，呼び出されたシステムコールの数は 969 個，未学習の動作パターンありの場合ではそのうち 3 個の呼び出しで，システムコール呼び出し順規則に一致しなかった．なお，システムコール呼び出し規則に一致しなかった動作パターンはすべて，ユーザ関数実行順規則とスタックリスト確認規則との比較で，正常な動作であるとが確認された．

表 1 の実験結果から動作 1 について，侵入検知システムによる監視時の実行時間は約 1.05 倍から 1.08 倍である．よって，実行時間の増加の割合は無視できる程度である．しかし動作 2 については，侵入検知システムによる監視時の実行時間は約 3.83 倍から約 4.32 倍となり，動作 1 の場合と比べて実行時間の増加の割合は大きい．これは，単位時間あたりに呼び出されるシステムコールの数が，動作 2 は動作 1 に比べて 3.5 倍以上も大きいため，スタックのトレースや，規則との比較に要した時間の割合が多かったためだと考えられる．

4.3　規則のサイズ

静的リンクによりコンパイルされた wc の実行ファイルが約 563KB であったのに対して，今回作成した 3 種類の規則のサイズは以下のようになった．

- 呼び出し順規則のサイズが約 1KB
- ユーザ定義関数実行順規則のサイズが約 4.5KB
- スタックリスト確認規則が約 61KB

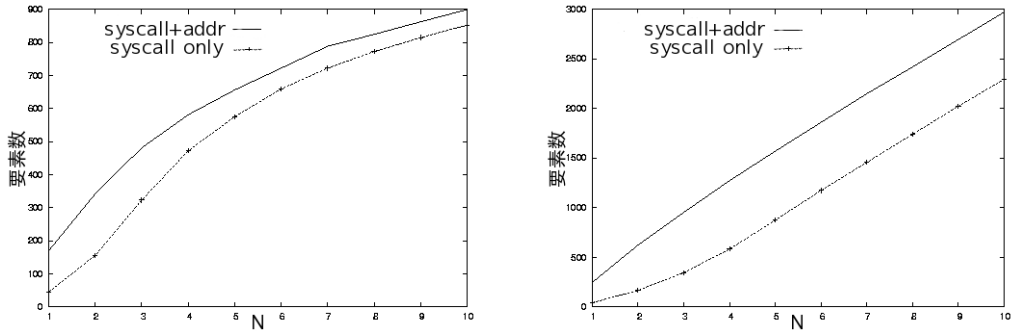
3 つの規則は合計は約 66.1KB となり，監視対象である wc の 12%の大きさになった．

共有ライブラリを使用する場合，wc の実行ファイルのサイズは約 49KB であり，静的リンクによる実行ファイルより小さく，規則サイズの大きさが目立つことになる．しかし，共有ライブラリを使用した場合には，仮想メモリ上でのライブラリ関数のアドレスは監視対象プログラムによらず一定であることから，スタックリスト確認規則 (約 61KB) は侵入検知プログラムに最初から組み込むことができ，実行ファイルごとに管理する必要がない．よって，残りの 5.5KB の規則を実行ファイルとともに管理するだけでよい．この場合，2 つの規則は wc のサイズの約 11%となる．どちらの場合においても，規則のサイズは実行ファイルの 1 割程度に抑えることができるため，無視できる程度である．

4.4　実行位置情報を用いた学習による影響

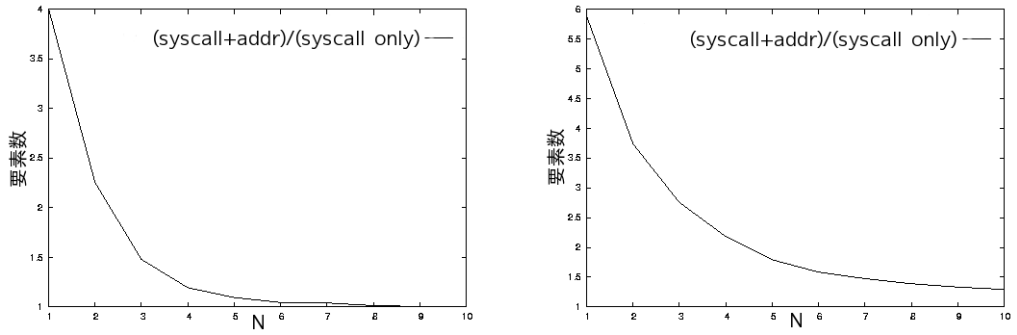
アプリケーションを実行したときに呼び出されるシステムコールについて，呼び出し順のみと，システムコールとユーザ定義関数の実行位置情報の組の呼び出し順のそれぞれについて，N-gram の要素数を調べた．対象としたアプリケーションは firefox と emacs の 2 つである．firefox は 10 分間ウェブブラウジングや各種操作をして，呼び出されたシステムコールとその実行位置情報を収集した．emacs はテキストファイルのオープンや，各種コマンドを用いての編集作業をしたときのシステムコールとその実行位置情報を収集した．図 4(a) 及び図 4(b) は firefox と emacs を，それぞれ実行したときの呼び出されたシステムコールとユーザ定義関数の実行位置情報から，N-gram($N = 0 \sim 10$) により生成された要素数を示している．また，図 5(a) 及び図 5(b) は firefox と emacs をそれぞれ実行したときにおいて，システムコールとユーザ定義関数の実行位置情報の組を，システムコールの呼び出し順のみと比較し，N-gram の要素数の倍率を示している．なお，アプリケーション firefox の実行時には 1878 回，emacs の実行時には 57533 回のシステムコールが呼び出しがあった．

システムコールの呼び出し順のみの場合は，実行位置情報も用いた場合に比べて要素数が少ないので，前者の方が同じ要素が現れやすいと言える．また，倍率が大きいことは，多数の実行位置が異なるシステムコール呼び出し列が，1 つのシステムコール呼び出し列として取



(a) firefox (b) emacs

図 4 N-gram の N とその要素数の関係



(a) firefox (b) emacs

図 5 N-gram の N とその要素数の倍率

りまとめられていることを意味している．これは，攻撃コードが正常な呼び出しパターンを偽装しやすくなっていることを意味している．つまり，システムコールの呼び出し位置を学習に加えることで，要素の重複を減少させている．これにより，攻撃者の偽装をより困難にしている．

4.5 実験結果の考察

実験結果から，システムコールの呼び出し情報のみに基づく N-gram では，異なる実行状況であっても同一のシステムコールの呼び出し順が現れることが多く発生していることが分かった．システムコールの番号とユーザ定義関数の実行位置情報を組み合わせることで規則の冗長性が取り除かれ，false negative の少ない規則が作成できると考えられる．また，それによりプログラムの実行の流れを十分に考慮した規則が作成可能である．

しかしシステムコール番号とプログラムの実行位置の情報を組にした要素を用いることは要素の種類の増加につながり，規則のサイズの増大や false positive の増加につながってしまう．そこで本研究では静的解析により作成された規則を組み合わせることで未学習の動作による false positive と異常動作の判断を実現できた．

5. まとめと今後の課題

正常な実行時パターンの学習により作成した規則と，実行ファイルの解析ににより作成した規則を用いて，プログラムの動作を確認しながら実行する検知システムを提案した．システムコール番号と実行位置情報の組の呼び出し列から N-gram 法を用いて特徴化する手法を提案し，その有用性を述べた．また，規則のサイズについては実行ファイルに比べて十分に小さいことを確認した．

しかし，プログラムの動作によっては監視したことによる実行時間のオーバヘッドが大きくなってしまうため，今後の改良が必要である．

今回作成した静的解析により規則を作成するプログラムでは，関数ポインタや long jump の対応ができてない．また，静的にリンクされた実行ファイルを解析対象としており，共有ライブラリを使用した実行ファイルに対しても同様に解析できるように改良する計画である．

また，システムコールの呼び出し順を表す規則は，N-gram 法を用いて作成したが，静的解析により規則を作成した場合のほうが誤検出が少なくなることが期待できる．しかし，冗長な規則になってしまうためおそれもあるため，false negative の発生率や，規則サイズの面か

ら有効性を検討していく．

今後，提案システムでどの程度の検知精度が得られるか，脆弱性が知られているプログラムに適用して検証していく予定である．

参 考 文 献

1) S.Forrest, S.A.Hofmeyr, A.Somayaji and T.A.Longstaff: A Sense of Self for Unix Processes, *Proc. 1996 IEEE Symposium on Security and Privacy*, Oakland, USA, pp.120–128 (1996).
2) D.Wagner and P.Soto: Mimicry Attacks on Host-Based Intrusion Detection Systems, *Proc. 9^{th} ACM Conference on Computer and Communications Security* (2002).
3) D.Gao, M.K.Reiter and D.Song: On Gray-Box Program Tracking for Anomaly Detection, *Proc. 13^{TH} USENIX Security Symposium* (2004).
4) R.Sekar, M.Bendre, D.Dhurjati and P.Bollineni: A Fast Automaton-Based Method for Detecting Anomolous Program Behaviors, *Proc. 2001 IEEE Symposium on Security and Privacy*, Oakland, CA, pp.144–155 (2001).
5) E.Eskin, W.Lee and S.J.Stolfo: Modeling System Calls for Intrusion Detection with Dynamic Window Sizes, *Proc. DARPA Information Survivability Conference and Exposition(DISCEX 2001)*, Anaheim, USA, pp.165–175 (2001).
6) C.Warrender, S.Forrest, B.Pearlmutter and B.Pearlmutter: Detecting Intrutions Using System Call: Alternative Data Models, *Proc. 2001 IEEE Symposium on Security and Privacy*, Oakland, pp.156–168 (2001).
7) nad D.Dean, D.: Interusion Detection via Static Analysis, *IEEE Symposium on Security and Privacy*, pp.144–155 (2001).
8) 安部 洋, 大山恵弘, 岡 端起, 加藤和彦：”静的解析に基づく侵入検知システムの最適化”，情報処理学会論文誌 Vol. 45, No. SIG 3(ACS 5), pp.11-20 (2004.3).
9) H.H.Feng, O.M.Kolesnikov, P.Fogla, W.Lee and W.Gong: Anomaly Detectioin Using Call Stack Information, IEEE Symposium on Security and Privacy, Berkeley, CA, pp. 62–77 (2003).