System Service 監視による Windows 向け異常検知システム機構

島 本 大 輔[†] 大 山 恵 弘[††] 米 澤 明 憲[†]

近年，不正なプログラムによる攻撃は極めて高度化している．多相型ウィルスや新種の攻撃コードなどによるいくつかの攻撃は，データのバイト列を攻撃のシグネチャと単純にマッチングする方式によるセキュリティシステムでは検知できないことがある．このような攻撃を検知するための有効な対策の一つに，プログラムの動作の監視による異常検知がある．本論文では，Windows の System Service の監視による異常検知方式を提案する．提案方式では，まず，アプリケーションの正常な動作を System Service 呼び出し動作のプロファイルから特徴化する．具体的には，System Service 呼び出しの N-gram 集合を生成し，それを正常な動作を表現するデータベースとして用いる．そして，監視対象のプログラムの動作をそのデータベースと比較することにより異常を検知する．我々は提案方式に基づく異常検知システムを実装し，現実的なアプリケーションを用いて実験を行った．実験では，特徴化に用いられるデータベースのサイズや異常を検知する能力について評価を行った．

Detecting Anomalies on Windows by Monitoring System Services

Daisuke Shimamoto,[†] Yoshihiro Oyama [†]
and Akinori Yonezawa[†]

In recent years, attacks by malicious programs are becoming highly sophisticated. Some new exploits and polymorphic viruses can evade the detection of security systems which depend on simple matching of byte sequences. An effective countermeasure against this kind of attacks is anomaly detection by monitoring the behavior of programs. In this paper, we propose an anomaly detection method that monitors System Services on Windows operating systems. The proposed method first characterizes the normal behavior of an application by using a profile of System Service calls. Specifically, it creates N-grams of System Service calls and utilizes it as a database representing the normal behavior. Then, it detects anomalies by comparing the behavior of monitored programs with the database. We implemented an anomaly detection system based on the proposed method and conducted experiments using realistic applications. Through the experiments, we have evaluated the size of database for characterization and the ability to detect anomalies.

1. は じ め に

情報システムに対するセキュリティ上の脅威は依然として大きい．ウィルス，ワーム，トロイの木馬などの不正なプログラムは数，種類共に増加している．また，プログラムのセキュリティホールを突くための多くの攻撃コードがインターネット上に公開されており，計算機に関して高いスキルを持たない者でも十分強力な攻撃を行うことができる状況になっている．

不正なプログラムへの対策として，ウィルス検出ソフトウェアの利用や，セキュリティホールを塞ぐパッチの適用が広く行われている．しかし，これらの対策を施しても防止が困難な攻撃が存在する．現在のウィルス検出ソフトウェアの多くは，不正なプログラムを特徴づけるパターン（シグネチャ）と，ファイルや通信のバイト列とのマッチングにより，不正なプログラムを検出している．この方法には，シグネチャが準備されていない新種ウィルスや，感染に伴いバイト列が変化する多相型ウィルスの検出が難しいという欠点が存在する．パッチを適用する方法にも，パッチが配布されるまでの間に行われる攻撃（ゼロデイ攻撃）が成功してしまうという問題が存在する[1]．

上記の攻撃を防止するための有効な対策の一つは，異常ベースの侵入検知（異常検知）を利用することである．異常検知は，シグネチャを用いる方法とは対照的に，正常なデータや動作を特徴づける規則を持ち，

† 東京大学大学院情報理工学系研究科コンピュータ科学専攻
University of Tokyo
†† 電気通信大学情報工学科ソフトウェア学講座
Univesity of Electro-Communications

それに合わないデータや動作を異常と判定する方式である．異常検知は，未知の攻撃や自身のコードを変化させる攻撃を検知しやすいという利点を持つ．

このような研究は UNIX 系 OS におけるシステムコールを対象に活発に行われており，高い有効性を示す結果も多く示されている．この方式では，たとえば，検査対象のプログラムが呼び出すシステムコール列が，正常なプログラムが守るべき規則に沿っているかどうかを検査する．これまで，この方式は主に UNIX 系 OS を対象に研究されてきた．その結果，この方式を Windows などの非 UNIX 系 OS に適用するための要素技術や，非 UNIX 系 OS 上における有効性は，十分に明らかになっていない．

本論文では，プログラムの動的な動作の監視による Windows 向け異常検知方式およびそれを実現するシステムを提案する．対象とする OS は NT 系列の Windows である．監視する動作は，システムコールに相当する，Windows の System Service 呼び出しである．提案システムは，まず，アプリケーションの正常な実行における System Service の呼び出し列を収集，解析し，正常な実行を特徴づける動作のデータベースを作成する．次に，動作がその特徴に従っているかどうかを検査しながらアプリケーションを実行する．正常な実行の特徴づけには N-gram 法[2] を用いる．System Service 呼び出しの監視は，デバイスドライバを導入し，x86 アーキテクチャの SYSENTER の動作を変えることによって実現している．我々は提案システムを実装し，有効性を評価するための様々な測定結果を得た．その結果，N-gram 法によって OS サービスの呼び出しを特徴化する異常検知方式は Windows においても有効であることがわかった．本研究は，我々の知る限り，OS サービスの呼び出し列を N-gram 法で特徴化する異常検知を Windows に適用した最初の研究である．

提案システムの特長を以下にまとめる．

- 異常な実行ではなく正常な実行を特徴化するので，未知の攻撃を検知することができる．
- プログラムのバイト列ではなく動的な振る舞いを特徴化するので，自身のバイト列を変化させるコードによる攻撃を検知することができる．
- カーネル内のデバイスドライバが System Service の監視を行うので，ユーザプログラムが監視を逃れることが難しい．
- System Service という低いレベルで監視を行っているため，任意のユーザプログラムの異常検知に利用できる．たとえば，Win32 API を用いるアプリケーションも POSIX API を用いるアプリケーションも監視することができる．

本論文では 2 章で System Service について説明する． 3 章で我々の提案システムについて述べ，その実験結果を 4 章で示す． 5 章で関連研究について述べ，6 章で結論と今後の課題について説明する．

2. System Service

2.1 概　　要

System Service はプログラムが Windows のカーネルの機能を利用するための機構，もしくは関数群である．System Service を呼び出すと，プログラムの制御はユーザレベルからカーネルレベルに遷移する．System Service は，ファイル操作，ディレクトリ操作，スレッド操作，プロセス操作などの OS の基本的な機能を含む．加えて，System Service はレジストリ操作などの Windows 独自の機能も含んでおり，UNIX 系 OS のシステムコールとは大きく異なるものとなっている．また，その他に Windows XP 以降では，Window Manager（以下，WM） と Graphics Device Interface（以下，GDI） の関数群が System Service として加えられている[3] ．WM は UNIX 系 OS における X-Window の機能にあたり，GDI はペンやブラシなどのグラフィックを担当する機能である．このため，System Service の数は非常に多く，Windows XP Service Pack 2 ではその数は 991 に及ぶ．

我々の研究は System Service を監視するため，OS 上で動作する任意のユーザプログラムを監視することが可能である．通常はアプリケーションが System Service を直接呼び出すことはなく，Windows が提供しているサブシステムと呼ばれる，System Service より抽象度の高い API 群を利用する．これにより，アプリケーションの開発者が System Service を把握する必要はない．System Service の追加や System Service 番号の付け替えは頻繁に行われており，Service Pack などの細かいアップデートの際にも変更されている場合がある．サブシステムは System Service を呼び出すことにより，カーネルを操作する．Windows では，Windows の基本的な機能を提供する Win32 サブシステムの他に，16bit Windows 互換，POSIX 互換，OS/2 互換のためのサブシステムが用意されている．

2.2 動作の詳細

図 1 に System Service の動作を示す．ユーザプログラムは OS の機能を用いる際，通常は Win32 などのサブシステムを呼び出す．サブシステムは System Service を呼び出し，カーネルモードに制御を移す．

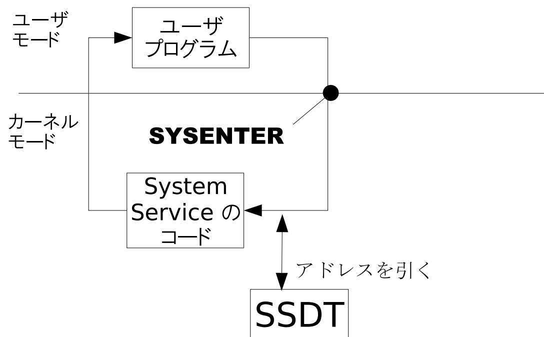

図 1 System Service の動作図
Fig. 1 System Service flow

カーネル内では，System Service 番号をインデックスとして，System Service Descriptor Table （SSDT）の要素を参照する．SSDT は UNIX 系 OS におけるシステムコールテーブルにあたるものであり，各 System Service のアドレスや引数の数などが記述されている．Windows のカーネルはこのテーブルから対応する System Service のアドレスを調べ，そのアドレスに制御を移す．

カーネルモードへの遷移は，Windows 2000 以前の OS では，ソフトウェア割り込みのための INT 命令☆により実現されているが，Windows XP では Pentium II から用意された SYSENTER 命令を利用している☆☆．

SYSENTER 命令はシステムコールの処理に特化された命令である．INT 命令を利用したシステムコールの処理では，ソフトウェア割り込みの割り込み関数に制御を移した後に，スタックポインタやセグメントポインタなどの値を変更し，対応するシステムコールのコードにジャンプするなどの処理を行う必要がある．SYSENTER 命令を利用すると，これらの処理を明示的に行う必要がない．SYSENTER 命令の実行により，カーネルモードへの遷移，セグメントポインタ，インストラクションポインタ，スタックポインタなどのレジスタの値が自動的に変更される．これらのレジスタの値は，CPU 内の特別なレジスタが持つ値に変更される．SYSENTER 命令により，システムコールや System Service の実行を高速化することができる．

☆ 具体的には int 0x2e である．Linux では，int 0x80 が使われている．

☆☆ Linux においても，カーネル 2.6 以降ではシステムコールの処理に SYSENTER 命令が採用されている．

3. 提案システム

3.1 基本的な枠組み

提案システムは，まず**学習モード**で実行され，後に**検知モード**で実行される．学習モードでは，監視したいアプリケーションが呼び出す System Service の列を記録する．学習モードにおける実行では異常や攻撃は存在しないと仮定する．アプリケーションを一定期間実行したら，学習モードでの実行を終了させる．そして，記録された列を元に，そのアプリケーションの正常な動作を特徴化するデータを生成する．動作の特徴化には N-gram 法 [2] を用いる．N-gram 法では，System Service 列の特徴を，その部分列（N-gram）の集合によって表現する．その集合には，ある与えられた数 N の長さの，連続する System Service 呼び出し列すべてが含まれる．たとえば，

A, B, C, B, C, B, A

という System Service 列が順に呼び出されたとする．この列から作られる N-gram の集合は，$N = 3$ とすると，

[A, B, C], [B, C, B], [C, B, C], [C, B, A]

の 4 要素からなる集合となる．この集合に含まれる部分列を正常な実行によるもの，含まれない部分列を異常な実行によるものとみなす．検知モードでは，System Service 呼び出しを監視し，最近の長さ N の部分列が，学習モードで生成した N-gram の集合に含まれるかどうかを検査する．含まれない場合には，正常な実行では観測されなかった動作が行われていると判断し，警報を発する．

現段階では，提案システムは，System Service の種別だけを利用し，System Service の引数や時刻情報は利用しない．ただし，今後，それらの情報も利用するようにシステムを拡張する可能性はある．本研究の狙いは，第一段階として，System Service の種別だけを用いる異常検知がどの程度効果的かを評価することにある．

3.2 System Service のフック

Windows では System Service 呼び出しを監視するための簡便な機構（例えば UNIX 系 OS における ptrace システムコールに類する機構）が，プログラマに提供されていない．System Service を監視するには，そのための機構を新たに構築する必要がある．

我々は System Service Interception という手法を利用して System Service のフックを実現する☆☆☆

☆☆☆ System Service Interception はセキュリティシステムだけ

我々は，カーネルモードで動作するデバイスドライバを導入し，SYSENTER 実行時のジャンプ先を書き換える手法によって System Service のフックを実現している．具体的には，ジャンプ先が格納されている SYSENTER_EIP_MSR というレジスタの値を，我々のデバイスドライバが提供するフック処理コードの先頭アドレスに書き換えている．この書き換えにより，アプリケーションが SYSENTER を実行すると，制御はフック処理コードに移る．なお，System Service の実行に INT 命令を用いる OS では，割り込み発生時のジャンプ先である割り込みベクタ内のアドレスを書き換えることにより，ほぼ同じ手法でフックを実現できる．本手法を用いると，監視対象ではないプロセスによる System Service 呼び出しもフックされるが，フック処理コードの先頭でプロセス ID を検査することにより，異常検知処理は監視対象プロセスのみに適用されるようにしている．

System Service Interception を実現するための別の手法としては，SSDT の書き換え（SSDT patching[5)]）がある．フックしたい System Service に相当するテーブルエントリを，フック処理コードのアドレスに書き換えれば，その System Service をフックすることができる．この手法は盛んに使われており，実装例やサンプルコードも多い．フックのオーバヘッドが，エントリを書き換えた System Service だけにしかかからないという利点もある．提案方式ではすべての System Service を監視する必要があるという理由から，我々は SYSENTER 実行時のジャンプ先を書き換える手法を採用した．

この手法では，カーネル内からの System Service 呼び出しを監視することはできない．本研究では，ユーザレベルで動作するプログラムの異常のみを扱うため，カーネル内からの System Service 呼び出しは考えない．そもそも我々のシステムで検知を意図しているのは，ユーザレベルで動作するプログラムが乗っ取られたり異常が発生した結果発生する異常な System Service 呼び出しである．

3.3 システムの構成

本研究のプログラムは（1） カーネル空間内にコードを挿入するためのデバイスドライバと，(2） そのデバイスドライバから情報を受け取り，学習・検知処理を行う GUI プログラムの 2 つからなる．

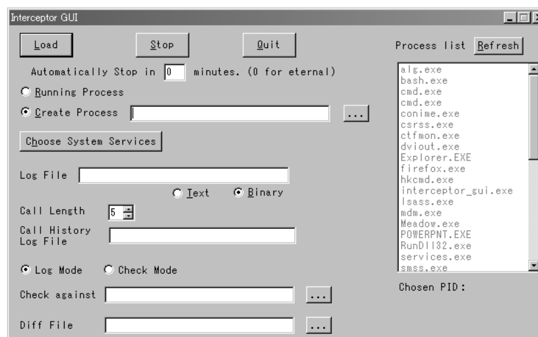

図 2 GUI プログラム
Fig. 2 The GUI program

3.3.1 デバイスドライバ

デバイスドライバの実行や停止は GUI プログラムから行う．SYSENTER の実行後のジャンプ先はこのデバイスドライバの中のフック処理コードである．デバイスドライバが実行されると，まず，監視対象のプロセスの ID をユーザーモードプログラムから受け取る．次に，元の SYSENTER_EIP_MSR の値を保存し，フック処理コードのアドレスを代わりにそのレジスタにセットする．フック処理コードにおいては，System Service の ID と呼び出し元のスレッド ID を記録する．

3.3.2 GUI プログラム

GUI プログラムはデバイスドライバと通信を行い，デバイスドライバを制御する．ある一定時間ごとに記録をデバイスドライバから受け取る．学習モードでは，この記録を受け取り，N-gram の集合を生成する．検知モードでは，呼び出された System Service が作る部分列が，学習モードで生成された N-gram 集合に含まれるかどうかを検査する．含まれない場合には異常が発生したとみなして警報を発する．GUI プログラムは **図 2** のようになっており，異常検知システムの動作を制御することができる．

本システム全体の動作図は **図 3** のようになる．

3.4 議　　論

異常検知のための監視を行う対象としては，System Service 以外では，Win32 API や DLL の呼び出しも考えられるが，我々は，以下の理由から System Service を監視する方式を採用した．まず，すべてのユーザプログラムは，あらゆるカーネルとのやりとりにおいて，System Service を通過する．System Service を迂回して OS のサービスを利用することは難しいため，侵入検知のための監視を行う場所として適している．さらに，System Service の呼び出しは SYSEN-

でなく，様々なプログラムで利用されている．不正なプログラムであるルートキットの実装にも利用されている．2005 年末，アメリカにおいて，SONY BMG の CCCD に含まれていたプログラムが System Service Interception を利用しており，ルートキットであると騒がれたのはそのためである．[4)]

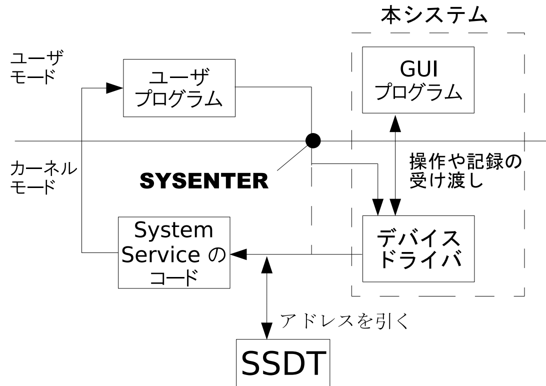

図 3 本システムの動作図
Fig. 3 The System Service flow with our system

TER 実行を伴うため，すべての System Service 呼び出しをフックすることが容易であるという利点もある．具体的には，System Service の各関数の仕様，個数，番号づけに依存しない形でフック機構が実装可能である．逆に欠点としては，本システムの実行後にルートキットなどにより，再度 SYSENTER のジャンプ先（SYSNTER_EIP_MSR）を書き換えられ，本システムが無効化される可能性がある．しかし，ルートキットによる上書きは SSDT patching など，他の手法も無効化することができる．解決策の 1 つとしては定期的にジャンプ先を確認し，改変されている場合は再度変更する，という方法が考えられる．SSDT patching においても同様のことが行えるが，この場合は SSDT のエントリをすべてをチェックしなくてはならないため，検査にかかるオーバーヘッドが大きくなる．この点においても本手法の方が SSDT patching より優れていると考えている．

提案方式ではプログラムが攻撃されている場合以外にも警報が出ることがある．警報が出る原因には，(1) N-gram 法が性来的に持つ分類誤り，(2) 学習モードで通過していない実行パス（レアパス）の通過の 2 つがある．前者による警報はユーザを混乱させる効果しかなく，できる限り少ないことが望ましい．一方，後者によって警報が出されることは，必ずしも欠点とは我々は考えていない．たとえ攻撃が存在しなくても，プログラムが稀な実行状態に入ったときに警報を出すことは，ユーザにとって都合が良い場合がある．たとえばデバイスの状態が普段と異なっているときや，資源不足によるエラーが発生しているときには，ユーザにとって警報が有用である場合がある．

4. 実　　験

本システムの有効性を調査するために実験を行った．まず，4.1 節において本システムの適用した場合のオーバヘッドを測定した．次に 4.2 節で N-gram の手法に有効と思われる N の値を調べるため，一般的なプログラムを対象に N-gram と branching factor を測定した．この結果を UNIX 系 OS との比較するために，Linux を対象として，同様の実験を行い，4.3 節にまとめた．最後に本システムの検知精度を測る指標として，4.4 節で false positive の値を調査した．なお，実験環境は 4.1 節，4.2 節，4.4 節においては Pentium M 1.3GHz，メモリ 512MB の PC に，OS は Windows XP SP1 である．4.3 節では Debian Linux 3.1（カーネル 2.4.27）をインストールした AMD Athlon 1.2GHz，メモリ 256MB の PC である．

4.1 オーバーヘッド

本システムのオーバーヘッドを測定した．実験としては，(1) マイクロベンチマークによるオーバーヘッドの測定と (2) 実用的な使用におけるオーバーヘッドの測定の 2 種類を行った．

まず，マイクロベンチマークのプログラムは，NtTestAlert() という System Service を 1 億回呼び，それに要する時間を計るというものである．結果を **表 1** に示す．我々の IDS を動作させているときは約 13.9%のオーバーヘッドが発生している．

さらに実用的なプログラムにおけるオーバーヘッドを計るために，Windows の explorer.exe を用いて，78.6MB のデータを含むフォルダに対して，“system” という単語を 4 回連続して検索した☆．その結果が **表 2** である．

1 回目の検索に時間がかかるのはディスクの遅延によるものであり，残りの 3 回はメモリ上のデータを検索しているため，速くなる．このメモリにキャッシュ

表 1 マイクロベンチマークの結果
Table 1 Result of the microbenchmark

	要した時間
IDS なし	60.9 秒
IDS 動作時	69.4 秒

☆ 4 回行っているのは，最初の検索の際にデータをメモリに乗せ，ディスクアクセスによる遅延の影響を小さくするためである．なお，元からデータがキャッシュされている可能性があるため，PC を再起動させた直後に，本システムを実行させていない状態で 4 回検索を行い，時間を計測した．さらに再起動させた後に本システムによる監視下で 4 回同様の計測を行った．

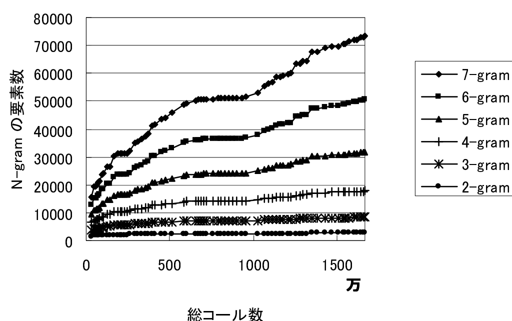

図 4 Firefox(Windows) の総コール数と N-gram の関係
Fig. 4 Relationship between the system calls made and the number of elements of the N-gram for Firefox (Windows)

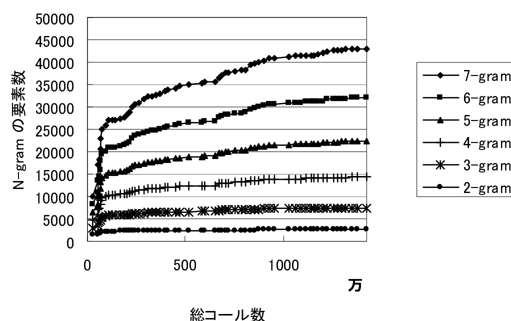

図 5 Impress(Windows) の総コール数と N-gram の関係
Fig. 5 Relationship between the system calls made and the number of elements of the N-gram for Impress (Windows)

されている場合の結果から，検索のオーバーヘッドは小さいことがわかる．

これら 2 つの実験より，本システムのオーバーヘッドはマイクロベンチマーク，実用的なプログラムにおいても小さいと言え，今後，より複雑な検知手法を用いて拡張することも十分可能と思われる．

4.2 N-gram と Branching Factor の測定

我々の異常検知システムを用い，2 種類のプログラムに対して System Service 列の N-gram を生成し，解析を行った．実験を行ったプログラムは以下の 2 つである．

(1) Firefox 1.5（以下，Firefox）
(2) OpenOffice.org 2.0 Impress（以下，Impress）

我々のシステムを学習モードに設定し，上記のプログラムを一定時間実行し，呼び出した System Service の列を記録する．具体的な操作として，Firefox では，Web ブラウジングやアップデートのチェックなどを行い，Impress では，新しくプレゼンテーションを作成し，文章の打ち込み，図の描画，アニメーションの作成，スライドショーの実行などの操作を行った．

図 4 と**図 5** は，Firefox と Impress をそれぞれ 70 分間，60 分間実行させたときに呼び出される System Service コールの N-gram（$N = 2 \sim 7$）の増加を示すグラフである．横軸は呼び出された System Service の総数，縦軸は N-gram の要素数となっている．

両方のグラフにおいて時間の経過とともに増加の傾きが緩くなるが，再度大きく増加する場合がある．これはユーザーが新しい動作を行ったタイミングと一致しており，System Service にグラフィックや GUI 関連のものが含まれていることによる影響が出ていると思われる．この際，$N = 6$ や $N = 7$ の場合にはこの増加量が非常に大きいことが分かる．これより，Forrest ら[6] の研究に見られた N を大きくすれば N-gram の数は増加しない，という結果は当てはまらないことが分かる．したがって，N が大きければ良い，という結果にはならない．

次にこの N-gram データから平均 branching factor を求めた．Branching factor とは，あるデータ列が存在するときに，その中から一定の長さの部分列を取り出し，その次に来る可能性がある要素の種類の数である．本研究の場合には，$N = 3$ のとき，要素数が 2 である特定の部分列を取り出し，その部分列の次に呼ばれ得る種類の数が branching factor になる．これをすべての部分列に対して求め，部分列の種類の数で割った平均が，$N = 3$ における平均 branching factor である．この値が小さいほど，長さ $N - 1$ の部分列の次に呼ばれ得る System Service の数が限られる．つまり，プログラムが攻撃され異常動作を始めた場合には，過去の記録に存在しない流れで System Service を呼び出す可能性が高くなる．したがって，平均 branching factor が小さいほど，高精度に異常検知を行うことができる． **表 3** が平均 branching factor を求めたものになる．

この表より，$N = 3$ 以上では平均 branching factor が約 3 か，それを下回っている．この結果から，$N = 3$ ないし $N = 4$ 程度の N-gram 学習データを用いて高い精度の異常検知が可能になると考えられる．

表 2 Explorer を用いた検索に要した時間
Table 2 Time it took for the explorer to finish searching

	1 回目	2 回目	3 回目	4 回目
IDS なし	1 分 27 秒	13 秒	13 秒	13 秒
IDS 動作時	1 分 39 秒	19 秒	14 秒	13 秒

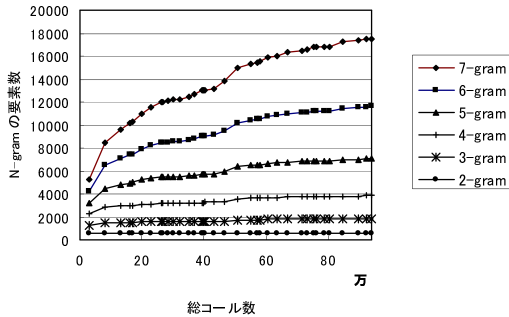

図 6 Firefox(Linux) の総コール数と N-gram の関係
Fig. 6 Relationship between the system calls made and the number of elements of the N-gram for Firefox (Linux)

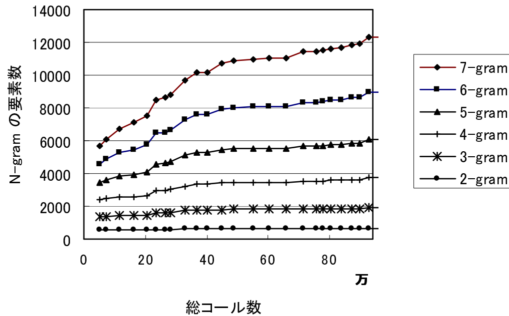

図 7 Impress(Linux) の総コール数と N-gram の関係
Fig. 7 Relationship between the system calls made and the number of elements of the N-gram for Impress (Linux)

4.3 Linux との比較

UNIX 系 OS と比較するために，4.2 節と同様の測定を UNIX 系 OS のシステムコールに対して行った．

実験対象のプログラムは 4.2 節と同じ Firefox と Impress の Linux 版である．システムコール列は strace を用いて取得した．それぞれの N-gram をグラフにしたものが **図 6** と **図 7** になり，**表 4** が branching factor の結果になる．

この結果から Linux においても，N-gram は N が大きい場合でも N-gram の要素数が増え続けている．

表 3 平均 branching factor（Windows）
Table 3 Average branching factor for Windows

	N					
	2	3	4	5	6	7
Firefox	10.25	3.10	2.13	1.79	1.59	1.45
Impress	10.86	2.93	1.92	1.57	1.43	1.34

また，$N = 3$ のおいて， branching factor も Windows と同様に 3 か，それ以下である．このことから，Windows と Linux で同程度な N の値を異常検知に用いることが可能になると思われる．なお，Linux におけるオーバーヘッドについては Somyaji らの研究[7] より，単純にシステムコールを呼び続けるマイクロベンチーマークのオーバーヘッドが 180 ～ 400% ほどであり，実際のアプリケーションではカーネルのビルドが 3.6%，”find / -print” のコマンド実行が 9.9% となっている．Somyaji らの実験では Linux のカーネルをソースコードレベルで変更しているが，本研究のように Windows のカーネルはソースコードレベルから変更できないことを考慮に入れると，本システムのオーバーヘッドはこれらの結果と比べ，遜色がないと考えている．

4.4 False Positive の評価

False positive とは，正常な動作を誤って異常な動作と判定してしまったデータのことである．したがって，セキュリティ対策ソフトにおいては，この値が小さいほど検知精度が高いということになる．本システムにおける false positive を評価する実験を行った． 4.2 節の際に得た N-gram を学習データとし，2 つのプログラムをそれぞれ 30 分ずつ実行させ，実験データとなる N-gram を生成する．この学習データと実験データの N-gram を比較し，積集合の要素数を求める．実験データのうち，積集合に含まれていない N-gram の要素が占める割合が false positive の割合となる．結果を **表 5** にまとめた．

N の値が大きくなるにつれ，false positive の割合

表 4 平均 branching factor（Linux）
Table 4 Average branching factor for Linux

	N					
	2	3	4	5	6	7
Firefox	7.81	2.92	2.13	1.84	1.65	1.51
Impress	8.40	3.07	2.00	1.66	1.51	1.41

表 5 実験データ内の false positive の割合
Table 5 Rate of false positives against the experiment data

	N-gram			
	1	2	3	4
Firefox	0.40%	4.99%	11.20%	17.45%
Impress	18.57%	33.89%	37.59%	40.26%

	N-gram		
	5	6	7
Firefox	23.36%	28.68%	33.31%
Impress	41.97%	43.06%	44.06%

が大きくなるため，異常検知には向いていないことがわかる．さらに，Impress の場合は，小さい N においても false positive が高い値となっている．つまり，これらの値を直接異常検知に適用することは望ましくなく，Warrender らの研究[8] のように一定の false positive をまとめた上で，閾値を用いる手法などを適用する必要があると思われる．また Impress についてはより学習させる必要もある可能性があり，今後詳しく調査していく予定である．

5. 関連研究

OS サービスを呼び出す挙動の学習による侵入検知方式は，主に UNIX 系 OS におけるシステムコールを対象に研究されてきた．Forrest らによる計算機免疫系 [2] は，システムコール列の N-gram を用いる侵入検知を提案した先駆的研究である．その後，他の情報や異なるアルゴリズムを用いる方式が多数提案された．例えば，有限オートマトンを用いる方式[9],[10]，システムコール引数を用いる方式[11]，スタック情報を用いる方式[12],[13]，ライブラリ呼び出しの情報を用いる方式[14]，可変長の N-gram を用いる方式[15],[16] などが提案された．我々の知る限り，本方式に関する過去のすべての研究は UNIX 系 OS を対象にしており，本方式を非 UNIX 系 OS 上で実現するための技術およびその有効性は明らかではなかった．本研究は，OS サービスの呼び出し挙動を N-gram で特徴化する侵入検知方式を非 UNIX 系 OS に適用した最初の研究である．

文献[17] の研究は，Windows レジストリアクセスを監視する侵入検知システムを提案している．提案システムは，正常時における Windows レジストリアクセスを学習し，それに合わないアクセスを異常と判定する．本研究の方式は System Service を監視するので，レジストリアクセスを伴わない攻撃も検出することができる．

Windows における重要な関数への呼び出しをフックするために，様々な手法が提案されている．まず，3.3.1 節で述べたように，カーネルドライバの導入により System Service をフックする手法として，SSDT の書き換え[5] と，SYSENTER 実行時のジャンプ先の書き換えがある．前者は様々な用途に広く用いられている．本研究では，安全性と実装の容易さなどの面から後者を採用したが，前者を用いて本研究のシステムを実装することも可能である．

重要な関数への呼び出しをユーザレベル機構のみを用いてフックする手法も提案されている．Detours[18] はプロセスのコード領域にパッチを当てることにより Win32 API のフックを実現する．vOS[19] は，アプリケーションコード内の import table を書き換えて DLL を切り替えることにより，Win32 API のフックを実現する．Mediating Connectors[20] は，共有ライブラリのコードにパッチを当てることにより共有ライブラリ呼び出しをフックする．これらの手法は，セキュリティ用途への適用という観点からは，悪意のコードが直接 SYSENTER を呼び出すことによりフックをバイパスできるという欠点を持つ．これらの手法を用いて侵入検知システムを構築するには，バイパスを防ぐための拡張が必要になる．

SSDT の書き換えを用いて System Service を監視するセキュリティシステムがいくつか提案されている．WHIPS[21] は，各 System Service の実行を許可するか禁止するかについてのポリシーにもとづいて，System Service の実行を制御する．ポリシーでは，System Service の種類と引数などを指示する．WHIPS では，OS に詳しいユーザが手間をかけてポリシーを作成しなければならないという問題がある．本研究のシステムでは，プログラムの挙動の学習により，正常な実行と異常な実行を分ける規則を自動的に生成することができる．Emu[22] では，System Service の監視をプログラムの実行制御に利用している．各ユーザには，実行してよいプログラムのリストが割り当てられる．Emu は ZwCreateProcess をフックすることにより，リストに含まれていないプログラムの実行を拒否する．Emu は，悪意による ZwCreateProcess の実行の防止に特化されているが，本研究のシステムはより広い範囲の侵入動作を検知できる．

未知の攻撃を検知する方式としては，プログラムコードの解析やエミュレーション実行を利用するものも提案されている．MEF[23] はベイズ分類を用いて，メールに添付されたファイルが悪意コードかどうかを推定するシステムである．SAFE[24] は，悪意のコードパターンをオートマトンで表現することにより，難読化処理に影響されにくい形で悪意コードを検出するコード解析器である．Symantec 社の Bloodhound[25] は，ウィルスかどうかわからないコードをエミュレーション実行して，その過程で得られる実行イメージや挙動を検査する．検査結果から，そのコードがウィルスかどうかを判断する．エミュレーション実行により，暗号化ウィルスに復号処理を実行させることなどができ，単純なバイト列のマッチングでは検知できないウィルスを検知できる．どのシステムも，本研究のシステムとは目的が異なり，ユーザが利用中のアプリ

ケーションの挙動を監視して異常を検知するものではない．

6. まとめと今後の課題

Windows における System Service の監視によって異常検知を行うシステムの設計と実装について述べた．我々のシステムでは N-gram 法を用いて正常な実行における System Service 呼び出し動作を特徴化し，その特徴に従うかどうかによって正常な実行と異常な実行を区別する．実験による評価の結果，効果的な異常検知が実現可能であることを示す測定結果が得られた．我々は長期的には，広範囲の OS に適用可能な，一般性のある異常検知方式を構築したいと考えている．本研究において，既存の多くの研究と異なり，UNIX 系以外の OS を対象に異常検知システムの開発と評価を行ったことは，そのための第一歩として意味を持つと考えている．

今後の課題としては，第一には，実際のウィルスや攻撃コードを用いて，さらなる評価を行いたい．Branching factor の評価により，攻撃を検知する能力は既にある程度見積もられていると考えるが，実験によって，さらに強い形で提案方式の有効性を実証していきたい．第二には，System Service の種別以外の情報も用いるような拡張を施すことにより，検知の精度を高めたい．

たとえばシステムコールと System Service の個数は全く異なるが，そのような相違に応じ，適応的に良いパラメタ設定を行って異常検知を行うようなシステムを構築したい．

5 章で述べたように，UNIX 系 OS に対する研究では，システムコール種別以外にも様々な情報を用いる手法が提案されている．まずはそれらの中で有望と考えられるものを Windows に適用することを考えたい．

System Service の層を用いて効果的な異常検知ができることを示した．我々はすでに System Service フックおよび監視のためのインフラを作ったので，今後，その上に様々な侵入検知のアルゴリズムを実装していくことができる．

参 考 文 献

1) eWeek: IE Under Attack: Microsoft Ponders Emergency Patch (2006). `http://www.eweek.com/article2/0,1895,1942566,00.asp`.
2) Forrest, S., Hofmeyr, S. A., Somayaji, A. and Longstaff, T. A.: A Sense of Self for Unix Processes, *Proceedings of 1996 IEEE Symposium on Security and Privacy*, Oakland, USA, pp. 120–128 (1996).
3) Microsoft: *MS Windows NT Kernel-mode User and GDI White Paper*. `http://www.microsoft.com/technet/archive/ntwrkstn/evaluate/featfunc/kernelwp.mspx`.
4) News.com, C.: Sony's Rootkit Fiasco (2005). `http://news.com.com/Sonys+rootkit+fiasco/2009-1029_3-5961248.html`.
5) Russinovich, M. and Cogswell, B.: Windows NT System-Call Hooking, *Dr. Dobbs Journal* (1997).
6) Forrest, S., Hofmeyr, S. A., Somayaji, A. and Longstaff, T. A.: A Sense of Self for Unix Processes, *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, IEEE Computer Society Press, pp. 120–128 (1996).
7) Somayaji, A. and Forrest, S.: Automated Response Using System-Call Delays, *Proceedings of the 9th USENIX Security Symposium*, Denver, Colorado, USA (2000).
8) Warrender, C., Forrest, S. and Pearlmutter, B. A.: Detecting Intrusions using System Calls: Alternative Data Models, *IEEE Symposium on Security and Privacy*, pp. 133–145 (1999).
9) Sekar, R., Bendre, M. and Bollineni, P.: A Fast Automaton-Based Method for Detecting Anomolous Program Behaviors, *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, Oakland, CA, pp. 144–155 (2001).
10) Wagner, D. and Dean, D.: Intrusion Detection via Static Analysis, *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, Oakland, California, pp. 156–168 (2001).
11) Krügel, C., Mutz, D., Valeur, F. and Vigna, G.: On the Detection of Anomalous System Call Arguments, *Proceedings of the 8th European Symposium on Research in Computer Security (ESORICS 2003)*, LNCS, Vol. 2808, Gjovik, Norway, pp. 326–343 (2003).
12) Feng, H. H., Kolesnikov, O. M., Fogla, P., Lee, W. and Gong, W.: Anomaly Detection Using Call Stack Information, *Proceedings of 2003 IEEE Symposium on Security and Privacy*, Berkeley, CA, pp. 62–77 (2003).
13) 阿部洋丈, 大山恵弘, 岡瑞起, 米澤明憲: 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).
14) Jones, A. and Li, S.: Temporal Signatures for Intrusion Detection, *Proceedings of the 17th Annual Computer Security Applications Conference*, New Orleans (2001).
15) Marceau, C.: Characterizing the Behavior of a Program Using Multiple-Length N-grams, *Pro-*

ceedings of New Security Paradigms Workshop 2000 (NSPW 2000), Cork, Ireland, pp. 101–110 (2000).

16) Eskin, E., Lee, W. and Stolfo, S. J.: Modeling System Calls for Intrusion Detection with Dynamic Window Sizes, *Proceedings of DARPA Information Survivability Conference and Exposition (DISCEX 2001)*, Anaheim, USA, pp. 165–175 (2001).
17) Apap, F., Honig, A., Hershkop, S., Eskin, E. and Stolfo, S. J.: Detecting Malicious Software by Monitoring Anomalous Windows Registry Accesses, *Proceedings of the 5th International Workshop on the Recent Advances in Intrusion Detection (RAID 2002)*, LNCS, Vol. 2516, Zurich, Switzerland, pp. 36–53 (2002).
18) Hunt, G. and Brubacher, D.: Detours: Binary Interception of Win32 Functions, *Proceedings of the 3rd USENIX Windows NT Symposium*, Seattle (1999).
19) Boyd, T. and Dasgupta, P.: Process Migration: A Generalized Approach using a Virtualizing Operating System, *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS 2002)*, Vienna, pp. 385–392 (2002).
20) Balzer, R. M. and Goldman, N. M.: Mediating Connectors: A Non-ByPassable Process Wrapping Technology, *Proceedings of DARPA Information Survivability Conference and Exposition (DISCEX 2000*, Hilton Head, South Carolina, pp. 1361–1368 (2000).
21) Battistoni, R., Gabrielli, E. and Mancini, L. V.: A Host Intrusion Prevention System for Windows Operating Systems, *Proceedings of the 9th European Symposium On Research in Computer Security (ESORICS 2004)*, Sophia Antipolis, France, pp. 352–368 (2004).
22) Schmid, M., Hill, F., Ghosh, A. K. and Bloch, J. T.: Preventing the Execution of Unauthorized Win32 Applications, *Proceedings of DARPA Information Survivability Conference and Exposition (DISCEX '01)*, pp. 175–183 (2001).
23) Schultz, M. G., Eskin, E., Zadok, E., Bhattacharyya, M. and Stolfo, S.J.: MEF: Malicious Email Filter - A UNIX Mail Filter That Detects Malicious Windows Executables, *Proceedings of 2001 USENIX Annual Technical Conference, FREENIX Track*, Boston, USA, pp. 245–252 (2001).
24) Christodorescu, M. and Jha, S.: Static Analysis of Executables to Detect Malicious Patterns, *Proceedings of the 12th USENIX Security Symposium*, Washington DC, pp. 169–186 (2003).
25) Corporation, S.: Dynamic heuristic method for detecting computer viruses using decryption exploration and evaluation phases, US Patent 6,357,008 (2002).