Vol. 47 No. SIG 12(ACS 15) 情報処理学会論文誌：コンピューティングシステム Sep. 2006

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きわ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1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，種類ともに増加している．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，感染にともな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1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
Department of Computer Science, Graduate School of Information Science and Technology, the University of Tokyo
†† 電気通信大学情報工学科
Department of Information Science, the 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バイト列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研究されており，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し，正常な実行を特徴づける呼び出し列のデータを作成する．次に，System Service の呼び出しがその特徴に従って，検査しながらアプリケーションを実行する．正常な実行の特徴化には 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分かった．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 章で本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 の数は非常に多く，Windows XP Service Pack 2 ではその数が 991 に及ぶ．

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．たとえば，System Service の大きな特徴して，Windows XP 以降では，上記のほかに，Window Manager と Graphics Device Interface の関数群が System Service として加えられている[3)．Window Manager は UNIX 系 OS における X-Window の機能にあたり，Graphic Device Interface 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．

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，アプリケーションは 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．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．よって，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．Windows では，Windows の基本的な機能を提供する Win32 サブシステムのほかに，16 bit 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

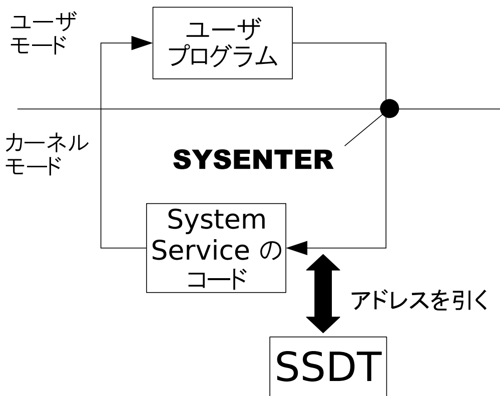

図 1　System Service の動作図
Fig. 1　Execution flow of System Service.

Service を呼び出し，カーネルモードに制御を移す．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，INT 命令によるソフトウェア割込み☆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，ソフトウェア割込みのハンドラ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．それに対して，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，スタックポインタ，セグメントポインタ，インストラクション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，記録された列をもとに，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．特徴化には N-gram 法[2) を用いる．すなわち，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見なす．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 1 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（たとえば 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．そこで，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☆．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，割込み発生時のジャンプ先である割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．この手法はさかんに使われている．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．しかし，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．

なお，本手法では，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が，提案システムの実現にあたっては，それは問題にならない．なぜなら，提案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の異常のみだからである．

3.3 システムの構成

本システムは，(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．

☆ System Service Interception はセキュリティシステムだけでなく，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]．

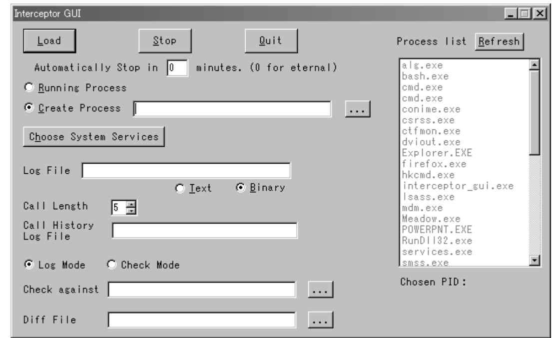

図 2　GUI プログラム
Fig. 2　The GUI program.

3.3.1 デバイスドライバ

デバイスドライバが実行されると，まず，監視対象のプロセスの ID を GUI 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．なお，このフック処理コードは，デバイスドライバの一部分として存在している．

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，System Service の ID とスレッド ID 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見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 SYSENTER 実行をともなうため，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．

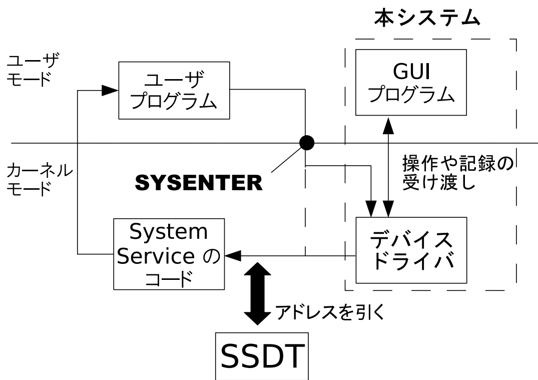

図 3　本システムの動作図
Fig. 3　Execution flow of System Service in our system.

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ふだん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 節では，本システムを適用した場合のオーバヘッドを測定した結果を述べる．次に，4.2 節で，実用的なプログラムを対象に N-gram の要素数と branching factor（後述）を測定した結果を示す．4.3 節で false positive の値を調査した結果を示す．なお，実験環境は 4.1 節，4.2 節，4.3 節においては Pentium M 1.3 GHz，メモリ 512 MB の PC に，OS は Windows XP SP1 である．

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
本システムなし	60.9 秒
本システム動作時	69.4 秒

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
本システムなし	1 分 27 秒	13 秒	13 秒	13 秒
本システム動作時	1 分 39 秒	19 秒	14 秒	13 秒

4.1　オーバヘッド

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

マイクロベンチマークを用いた実験は，NtTestAlert() という System Service を 1 億回呼び出すのに要する時間を計るものである．結果を**表 1** に示す．本システムを動作させているときは約 13.9%のオーバヘッドが発生している．このマイクロベンチマークでは System Service を呼び出す以外の処理は行わないため，オーバヘッドはある程度大きい．しかし，他のプログラムでは System Service 呼び出しの頻度はより低くなるため，オーバヘッドは 13.9%よりも小さくなると考えられる．

さらに，実用的なプログラムにおけるオーバヘッドを計るために，Windows の explorer.exe を用いて，78.6 MB 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，と我々は考えている．検索処理に対して本システムが加えたオーバヘッドは

☆ 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を実行させた状態で 4 回同様の計測を行った．

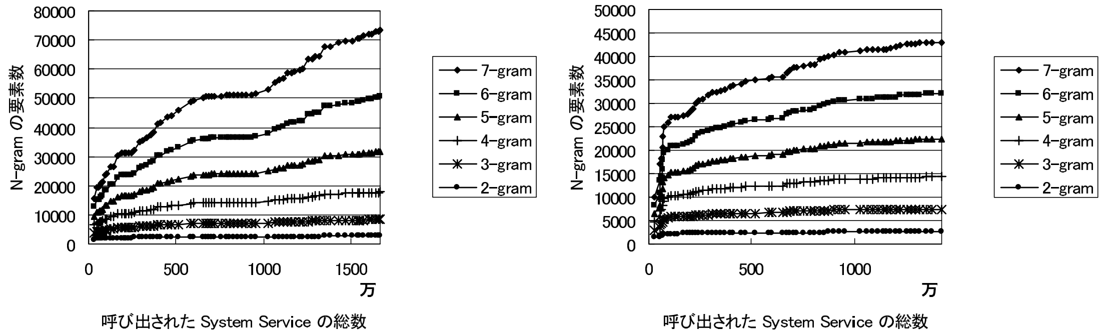

図 4　Windows における，呼び出された System Service の総数と N-gram 集合の関係（左図：Firefox，右図：Impress）
Fig. 4　Relationship between the number of System Service calls made and the number of elements in the N-gram set for Windows (left: Firefox, right: Impress).

13.7%以下であった．

なお，Linux におけるオーバヘッドについては Somayaji らの研究[6] では，同じシステムコールを呼び続けるマイクロベンチマークのオーバヘッドが，システムコールの種類によって異なるが，180〜400%ほどである．実際のアプリケーションでは，カーネルのビルドを行った場合のオーバヘッドが 3.6%，“find / -print” のコマンドを実行した場合が 9.9%という結果が出ている．本システムのオーバヘッドは Linux 上のシステムのそれに比べて，大きく増加してはいない．

4.2　N-gram の要素数と Branching Factor の測定

本システムを以下の 2 種類のプログラムに適用し，正常な実行を特徴化するデータを作成した．

(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た．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 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 の N-gram（$N = 2 \sim 7$）の要素数が増加する様子をまとめたグラフである．

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．これはユーザが新しい動作を行ったタイミングと一致している．この大きく増加する際，$N = 6$ 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．

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

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うる種類の数が 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うる 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．

UNIX 系 OS と比較するために，同様の測定を Linux のシステムコールに対して行った．実験には，Debian Linux 3.1（カーネル 2.4.27）をインストールした AMD Athlon 1.2 GHz，メモリ 256 MB の PC

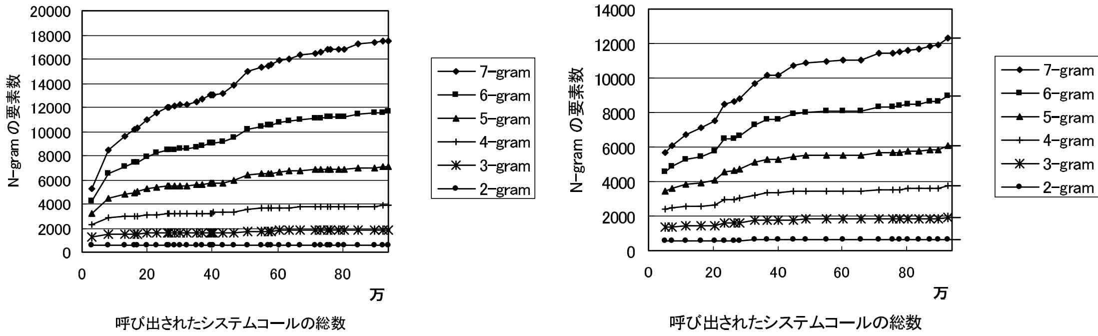

図 5　Linux における，呼び出されたシステムコールの総数と N-gram 集合の関係（左図：Firefox，右図：Impress）
Fig. 5　Relationship between the number of system calls made and the number of elements in the N-gram set for Linux (left: Firefox, right: Impress).

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	5	6	7
Firefox	0.40%	4.99%	11.20%	17.45%	23.36%	28.68%	33.31%
Impress	18.57%	33.89%	37.59%	40.26%	41.97%	43.06%	44.06%

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

を利用した．実験対象のプログラムは，同様に Firefox と Impress の Linux 版である．システムコール列は strace コマンドを用いて取得した．それぞれの N-gram の要素数をグラフにしたものが**図 5** 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．また，$N = 3$ のおいて，branching factor も Windows と同様に約 3 か，それ以下である．すなわち N-gram 要素数と branching factor に関しては，Linux と Windows で似た傾向を示した．

4.3　False Positive の評価

False positive とは，誤って異常と判定されてしまった正常な動作や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，この数が少な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．False positive の割合は $N = 3$ の段階で 10%を超えており，少なくない数の false positive が発生していることが分かる．この結果を考慮すると，学習データに含まれない System Service が 1 つ出現した段階で即異常と判定することは望ましくない可能性がある．Warrender らの研究[7] のように，学習データに含まれない N-gram が一定期間内にある閾値以上の個数観測されたら異常と判定するなどの方法を適用する必要があると思われる．

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．たとえば，有限オートマトンを用いる方式[8),9)]，システムコール引数を用いる方式[10)]，スタック情報を用いる方式[11),12)]，ライブラリ呼び出しの情報を用いる方式[13)]，可変長の N-gram を用いる方式[14),15)] 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 16) 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のパターンを学習し，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ともな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 項で述べたように，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[17] 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[18] 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[19] 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[20] 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に基づいて，System Service の実行を制御する．ポリシでは，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を作成しなければならないという問題がある．本システムでは，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[21] 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[22] 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[23] は，悪意コードの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[24] は，ウィルスかどうか分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．どのシステムも，本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．また Mori の研究[25] では，コードの静的な解析を行うことにより，未知のウィルスを検知を試みている．

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．本システムでは 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 1 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 1 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すで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 2 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．第 3 に，他の OS における実験を行い，OS に依存しない検知手法について考えていきたい．

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, *Proc. 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) CNet News.com: 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) Somayaji, A. and Forrest, S.: Automated Response Using System-Call Delays, *Proc. 9th USENIX Security Symposium*, Denver, Colorado, USA (2000).
7) 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).
8) 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, CA, pp.144–155 (2001).
9) Wagner, D. and Dean, D.: Intrusion Detection via Static Analysis, *Proc. 2001 IEEE Symposium on Security and Privacy*, Oakland, California, pp.156–168 (2001).
10) Krügel, C., Mutz, D., Valeur, F. and Vigna, G.: On the Detection of Anomalous System Call Arguments, *Proc. 8th European Symposium on Research in Computer Security* (*ESORICS 2003*), LNCS, Vol.2808, Gjovik, Norway, pp.326–343 (2003).
11) Feng, H.H., Kolesnikov, O.M., Fogla, P., Lee, W. and Gong, W.: Anomaly Detection Using Call Stack Information, *Proc. 2003 IEEE Symposium on Security and Privacy*, Berkeley, CA, pp.62–77 (2003).
12) 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：コンピューティングシステム，Vol.45, No.SIG 3 (ACS 5), pp.11–20 (2004).
13) Jones, A. and Li, S.: Temporal Signatures for Intrusion Detection, *Proc. 17th Annual Computer Security Applications Conference*, New Orleans (2001).
14) Marceau, C.: Characterizing the Behavior of a Program Using Multiple-Length N-grams, *Proc. New Security Paradigms Workshop 2000* (*NSPW 2000*), Cork, Ireland, pp.101–110 (2000).
15) Eskin, E., Lee, W. and Stolfo, S.J.: 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).
16) Apap, F., Honig, A., Hershkop, S., Eskin, E. and Stolfo, S.J.: Detecting Malicious Software by Monitoring Anomalous Windows Registry Accesses, *Proc. 5th International Workshop on the Recent Advances in Intrusion Detection* (*RAID 2002*), LNCS, Vol.2516, Zurich, Switzerland, pp.36–53 (2002).
17) Hunt, G. and Brubacher, D.: Detours: Binary Interception of Win32 Functions, *Proc. 3rd USENIX Windows NT Symposium*, Seattle (1999).
18) Boyd, T. and Dasgupta, P.: Process Migration: A Generalized Approach using a Virtualizing Operating System, *Proc. 22nd International Conference on Distributed Computing Systems* (*ICDCS 2002*), Vienna, pp.385–392 (2002).
19) Balzer, R.M. and Goldman, N.M.: Mediating Connectors: A Non-ByPassable Process Wrapping Technology, *Proc. DARPA Information Survivability Conference and Exposition* (*DISCEX 2000*), Hilton Head, South Carolina, pp.1361–1368 (2000).
20) Battistoni, R., Gabrielli, E. and Mancini, L.V.: A Host Intrusion Prevention System for Windows Operating Systems, *Proc. 9th European Symposium On Research in Computer Security* (*ESORICS 2004*), Sophia Antipolis, France, pp.352–368 (2004).
21) Schmid, M., Hill, F., Ghosh, A.K. and Bloch, J.T.: Preventing the Execution of Unauthorized Win32 Applications, *Proc. DARPA Information Survivability Conference and Exposition* (*DISCEX 2001*), pp.175–183 (2001).
22) 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, *Proc. 2001 USENIX Annual Technical Confer-*

ence, *FREENIX Track*, Boston, USA, pp.245–252 (2001).
23) Christodorescu, M. and Jha, S.: Static Analysis of Executables to Detect Malicious Patterns, *Proc. 12th USENIX Security Symposium*, Washington DC, pp.169–186 (2003).
24) Symantec Corporation: Dynamic heuristic method for detecting computer viruses using decryption exploration and evaluation phases, US Patent 6,357,008 (2002).
25) Mori, A.: Detecting Unknown Computer Viruses —A New Approach, *Proc. ISSS 2003* (2003).

(平成 18 年 1 月 27 日受付)
(平成 18 年 5 月 25 日採録)

島本　大輔
1982 年生．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修士 2 年．興味はシステムソフトウェア，コンピュータセキュリティ，並列分散処理.

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）．科学技術振興事業団研究員，東京大学大学院情報理工学系研究科助手を経て，現在，電気通信大学情報工学科助教授．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.

米澤　明憲（正会員）
1947 年生．MIT 計算機科学科博士課程修了（Ph.D.），MIT 計算機科学研究所および人工知能研究所において並列・分散計算モデルの研究に従事し，「並列オブジェクト」概念のパイオニアの 1 人．帰国後，東京工業大学を経て，1988 年より東京大学大学院コンピュータ科学専攻（当時理学部情報科学科）教授．米国計算機学会フェロー（ACM Fellow），ACM TOPLAS 副編集長，日本ソフトウェア科学会理事長，ドイツ国立情報科学技術研究所（GMD）科学顧問，政府情報科学技術委員会委員等を歴任．現在，（独）産業技術総合研究所情報セキュリティセンター副所長，東京大学情報基盤センター長.