Vol. 46 No. 8 情報処理学会論文誌 Aug. 2005

分岐命令処理フィルタを用いた不正プロセス実時間防御機構の構築

安藤類央[†] 武藤佳恭[††]

2001年度からの複合攻撃を行う不正コードの出現により，公的/私的機関を問わず基幹サーバのセキュリティは社会問題になりつつあり，不正プロセスの実時間での防御と，新種の攻撃方法への対応のタイムラグによって生じる被害についての技術的な解決策が求められている．本論文では，プロセッサのデバッグ/命令トレース機能を機軸とした，分岐命令処理フィルタを用いたリアルタイムの不正プロセスの制御システムの提案と実装評価を行う．同手法により，脆弱性を孕むソフトウェアのリビルドを行うことなく，不正なアクセスを受けたプロセスの検出と停止を実時間で行うことが可能になることが明らかになった．本論文ではシグニチャベースの方式に代えて，プロセスデバッグベースの方式を採用することで，従来手法では不可能であったソフトウェアの再構築の回避とバッファオーバフローを起こしたプロセスの実時間での停止という2つの防御要件を同時に達成することが可能であることを示す．提案システムの機能評価として，多形態型のバッファオーバフローを利用する攻撃を扱う．評価実験では，提案システム稼動時の負荷率と，関数呼び出し関数に応じた負荷率を測定した．また，防護対象にデータベースサーバのプロセスを設定し，トランザクションの内容に応じて，システムリソースの利用率を測定した．結果として，外部ネットワークに接続するサーバの処理内容に対して，適切な負荷で提案システムが稼動することが明らかになった．

Branch IP Filter for the Real-time Infected Process Nullification

RUO ANDO[†] and YOSHIYASU TAKEFUJI[††]

Since the emergence of malicious code loading blended attack in 2001, it has been required to prevent the malicious process on real-time and minimize the damage caused by delay time about updating vulnerable program to protect mission critical servers. In this paper, we introduce a branch IP filter, based on debug and instruction trace technology for real-time malicious process nullification. The proposal system, which relied on the concept of process debugging is applied for the prevention of buffer overflow, which makes it possible to nullify the infected process without rebuilding the application. Previously, no technology of stopping malicious process without recompiling source code or rebuilding software has been proposed. The branch IP based technique enables us to achieve real-time nullification of illegal execution and evading rebuilding software at the same time. In experiment, CPU utilization of detecting bufferoverflow, CPU time corresponding the number of calling function and protecting some operation of database server is measured and evaluated. As a result, proposal system could be applied for the servers with reasonable system resource utilization for the operation of servers connected to the Internet.

1. はじめに

ここ数年の各産業のインターネットの利用率の増加は著しい．情報化社会とは，一面ではIT産業が新規産業として位置付けられるということだけでなく，電子政府への取り組み，企業のERP，CRMなどの積極的な導入に象徴されるように，産業全体がインターネットという開放的な情報通信システムによって有機的に接合され，その連関を再編成していると解釈することができる．しかし同時に，軍や学術機関の研究から端を発したインターネットが民間の産業と生活に普及にしていくにつれて，これからの課題は情報通信システムのセキュリティをどのように確保していくかに重点がおかれつつある．

今世紀の初頭から，電子政府，電子商取引が本格的に実現しつつあることから，インターネットは次世代の社会インフラの基幹を支える技術になりつつある．情報通信産業内に限らず，企業は情報化を進めることでコストの削減を図ることが通例となってきている．これらの事柄に併行して，企業と社会インフラを担うソフトウェアの需要は飛躍的に伸びており，開発に求

† 慶應義塾大学大学院政策メディア研究科
Graduate school of media and governance, Keio University
†† 慶應義塾大学環境情報学部
Faculty of environmental information, Keio university

められるスピードも急激に上がってきている．そのため，十分な検証を行い，ソフトウェアバグが完全にない状態にして出荷するという目標が，非常に重要ではあるが，達成が困難になりつつあるのが現況である．

オンラインチケット，電子チェックイン業務，そしてATMなどの，ユーザが直接関与する機能を停止させた2003年1月Win32 SQLワームは，セキュリティ侵害が顧客や利用者に及んだという点で，不正コードが社会問題化した最初のケースであると指摘されている．このインシデントについて，リリースする前のソフトウェアのバグの完全な検査と，脆弱性が発見された際にパッチを当てる現状方式の問題点が露呈されたと見ることができる．

2001年からの複合攻撃を行うウィルス/ワームの出現は，ネットワークセキュリティが社会的な問題として取り上げられるようになった原因の1つだといえる．また，これらの不正コードは，現存するOSやアプリケーションを構築するC/C++に代表される厳密な型検査，バウンドチェックを行わない開発言語の性質に起因するバッファオーバフローを利用しているものが多い．

本論文では，以上の社会的背景とコンピュータセキュリティの現状をふまえて，バッファオーバフローにより発生する不正プロセスに対し，実時間で検出，停止を行い，かつ防御対象となるソフトウェアのリビルドを必要としないシステムを提案する．提案手法は，プロセッサのデバッグ/命令トレース機能を機軸とするため，オペレーティングシステムとは独立して実装でき，稼動が可能である．

提案システムの実装には，例外ハンドラの改良と，プロセス構造ルーチンを用いたデバッグポイントのメモリ空間へのロード時での挿入を用いることにより，防御対象となるプロセスが，自己をデバッグし，攻撃が成功した場合に，自己を停止することのできるAutomated debug属性を付与するところに特徴がある．

改良例外ハンドラは，プロセス構造ルーチンによって登録されるドライバコールバック関数によって呼び出される．また，Automated debug属性の有効化について，本論文では，自己デバッグにはIA-32プロセッサが発生する例外であるINT01H，停止にはINT03Hを利用した．INT01H，INT03HはIA（Intel architecture)-32の例外ベクタであり，INT01Hはデバッグ例外（debug）と呼ばれ，トラップまたはフォルトとして扱われる．INT03Hはブレークポイント（breakpoint）であり，トラップとして扱われる．通常はデバッガがこの命令を埋め込むが，提案システムでは攻撃を受けたプロセスの停止に同命令を用いた．

2. 関連研究

不正プロセスの検出には，静的に解析をする方法と，動的に検出する方法があり，前者は，対象となるコードがメモリにロードされる前に，データマイニングやヒューリスティックなどの手法を使って解析する方法で，後者はコードをメモリにロードし，シミュレーションあるいはランタイムで検出を行う手法である．

2.1 静的検出法

静的検出法は，プログラムを実行せずに検出が可能な方法であり，パターンマッチング法を中心として，比較方式とヒューリスティック方式に分かれる．

コンペア法/チェックサム法/インテグリティチェック法[1]は，未感染のプログラムの情報を格納しておき，監視対象のコードの変化を検出する．パターンマッチング法[2]は，感染コードの特徴的なバイトコードが，対象となるプログラムに存在するか検査する方式である．ヒューリスティック方式[3]は，感染後の振舞いを引き起こすと見なされやすいコードを解析することで，プログラムの異常を検出するものである．一般に，静的検出法は，ステルス技法の施されたコードや，ポリモーフィック/メタモーフィック化したコードに対しては，解析検出が困難であるという欠点が指摘されることが多い．

2.2 動的検出法

動的検出法は，ダイナミックヒューリスティック法とも呼ばれ，監視対象となるコードを実際に動作させ，あらかじめ定めておいたルールに適合する挙動を検出する方法であり，実行環境は実マシンあるいはエミュレータ両方の場合がある[4]．そのなかでも，バッファオーバフローを引き起こすプロセスを検出する方法としては，コンパイラベース，カーネルベース，そしてプロセッサ機能と併用する非実行スタック防御の3種類があげられる．

コンパイラベースの検出技術は，フレームポインタの隣接にカナリアと呼ばれる値を設置し，この値がバッファオーバフロー時に改変されることを利用して検出を行うものである．代表的なものにStackGuard[5]とPropolice[6),7)]があるが，前者は実コードで検出を行うのに対して，後者は中間言語において検出を行うことから，StackGuardがIntel^{TM} CPUに特化しているのに対し，Propoliceは，サポートするアーキテクチャに幅があり，また，引数やフレームポインタの書き換えにも対応できる．以上2つに代表されるコンパ

イラベースの防御策には，バッファオーバフローの脆弱性が発見されたアプリケーションやカーネルのリビルドを行わなければならない，また，バッファオーバフローの検出はできるがこれに対してのプロセスの停止などの措置はとれないなどの欠点が指摘されている．

同様に，コンパイラベースの手法として，メモリオブジェクトのトレーシング機能を利用しているのが，Bounds Checking[8] である．Bounds Checking ではすべてのメモリ上のオブジェクトはその利用がプログラムによって追跡できるようにコンパイラが拡張されている．これによって，理論的には，バグの種類にかかわらずメモリの不正利用を検出することができ，リターンアドレスの改ざん防止などの防御が可能になる反面，他の防御手法に比べると負荷が高く，脆弱性が発見されたアプリケーションやカーネルに対しては Bounds Check 用のライブラリをリンクし，アプリケーションを再コンパイルする必要がある．

Openwall[9),10)] はバッファオーバフローをカーネルの拡張で防御するもので，ユーザメモリエリアにおけるスタックの非実行化機能に特徴がある．カーネルベースの防御では，スタックのデータの実行を禁止することができ，バッファオーバフローが発生しても不正なコードを実行することができないようになっている．カーネルレベルで保護機能が発揮されるため，Openwall では，アプリケーションを再コンパイルする必要がなく，実行時の負荷が比較的小さいという特徴があるが，Openwall 自体のカーネルの再コンパイルは必須で，StackGuard 系の防御システムと同じくバッファオーバフローの検出はできても不正プロセスの停止などの措置はとれないことが課題となっている．

3. 検出対象の定義

ここでは，提案システムが有効な攻撃手法である多形態型のバッファオーバフローについて述べる．近年，ポリモーフィック[11)] やメタモーフィック[12)] といった従来のファイルスキャナを回避する手法が議論されることが多かったが，最近はリモートバッファーオーバフローのシグニチャ回避に，多形態化を利用する手法が，IDS（Intrusion Detection System）の運営上非常に重要な問題になってきている．

現在のオペレーティングシステムは，アセンブラより上の抽象レベルでの，C 言語に代表される高水準言語によって構成される．これらのシステムの基幹をなすテクニックは，関数と変数の利用であり，これによって制御を，本体とは別に局所的にまとめられ，繰り返し利用されることを想定して構成された手続き（関数）に移し，実行終了後に呼び出し元に返るという処理手順が可能になる．これにより，複雑かつ精巧な機能を持つソフトウェアの開発が可能になったが，現存するソフトウェアのほとんどは，厳密な型チェックを行わない C 言語によって構築されていることから，結果として今日のセキュリティ侵害の起因となるソフトウェアのバグはバッファオーバフローが大きな割合を占めている．

表 1 NIST データベースから脆弱性の分類
Table 1 Vulnerabilities list from NIST database.

	user	root
buffer overflow	1,269	520
format string	153	70
mata character	155	58
race condition	145	52
boundary error	254	32
dir traversal	309	17
xss	190	14
unknown	74	7

データソース: http://icat.nist.gov/icat.cfm

表 1 は，NIST の脆弱性データベースからの出典で，最近のセキュリティ侵害の原因となっているソフトウェアバグを分類集計したものである．このように，リモートからの権限奪取に使われる手段としては，バッファオーバフローが非常に多い割合を占める[13)]．また，深刻な被害をもたらすルート権限の奪取を引き起こすバグについてはバッファオーバフローが全体の 70%程度を占めている．このような現況に対応して，IDS や他の防御システムもバッファオーバフローに関するシグニチャを多く搭載している．

図 1 はバッファオーバフローによる不正なリターンアドレスの書き換えについて示したものである．

ここで，buffer は関数の外に置かれた定数，sfp は saved frame pointer で，主にデバッグ用途に参照される．return address は関数実行後に，instruction pointer に格納される関数呼び出し元のアドレスが格納されている．バッファは，データ型の 1 つ以上のインスタンスを保持するため，メモリ上に用意される連続したブロックのことである．バッファオーバフローは，あらかじめ設定された固定長のバッファに対して，不当な長さの入力を行うことで，バッファに割り当てられたメモリブロックの外にデータが置かれ，バッファに隣接するレジスタや変数の値が書き換わってしまう現象のことをいう[14)]．

図 1 に即すと，buffer にその長さ以上の文字列をコピーすることで，sfp と return address の値を，意図的に変更することが可能である．ここでは，バッファ

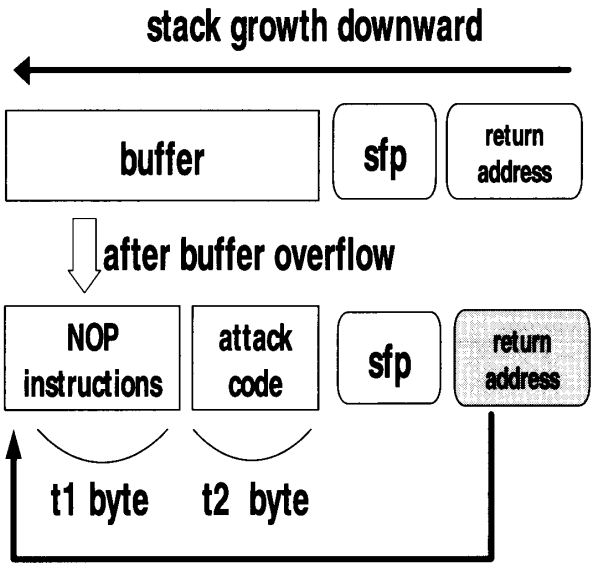


図 1 バッファオーバフロー
Fig. 1 Buffer overflow.

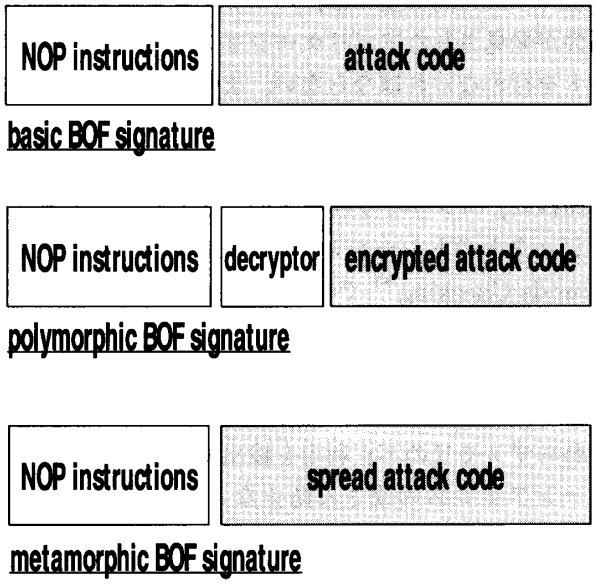


図 2 多形態型のバッファオーバフローのシグニチャ
Fig. 2 Signature of polymorphic/metamorphic buffer overflow.

オーバフローを起こすことで，NOP 命令文字列の先頭に instruction pointer が移動し，攻撃コードが実行される仕組みを示した．

一般に，攻撃の対象となるバッファの長さに則して，不正コードへのオフセットを推測することは非常に難しい．そのため，バッファオーバフローを用いた攻撃では不正アクセスを行うペイロードの先頭に，連続した NOP 命令文字列を付加して正確なオフセットの推測をしなくても，リターンアドレスの参照先が NOP 命令文字列のいずれかのアドレスになるようにすることで，攻撃を成功させる確率を上げる手法が通例となっている．

図 1 の下部は，オーバフローを起こした後に実行させる不正コードと，NOP 命令文字列の長さを示したものである．ここで，攻撃が成功する確率を $P\ (exploit)$ とすると，

$$P(exploit) = 1 - (f(t2) - f(t1)) \qquad (1)$$

となる．ここで，$f(t1)$ と $f(t2)$ は，$t1$，$t2$ に関して正比例する．つまり，確率を上げるためには，原則的には $t2$ はできるだけ短く，$t1$ は可能な限り長くする必要がある．そのため，ネットワークベースでの IDS (NIDS) におけるバッファオーバフロー検出のシグニチャには NOP 命令文字列を採用しているケースが多い．これに対して，NIDS のリモートバッファオーバフローのシグニチャ検出を回避するテクニックとして，ポリモーフィックバッファオーバフローという手法が提案されており，これは NOP 命令文字列に暗号化を施し，別の形態のコードにエンコードするというものである．ポリモーフィックとは，多形という意味で，もともとウィルスがホストベースでのファイルスキャンを回避するため，暗号化により自身のシグニチャを変形させる手法であったが，同手法の適用として，リモートからターゲットのマシンへバッファオーバフローを引き起こすペイロードに，このような変形を施して検知システムの回避を行う手法が提案されている[15]．

図 2 に，リモートバッファオーバフローを引き起こす 3 種類のシグニチャを示した．一般に，自己のシグニチャを改変することでストリングパターンマッチを回避する手法には，図 2 の下から 2 つに示した，メタモーフィックとポリモーフィックの 2 種類がある．現況では，リモート攻撃のシグニチャを改変するケースにおいては，両者をまとめてポリモーフィックと呼称され，議論されているが，本論文では，機能的に等価な攻撃コードを，複数の命令による異なる実装方法によって実現することをメタモーフィックバッファオーバフローとして，区別して扱う．この手法に対する提案手法の有効性については，評価実験の章で後述する．

4. 提案システム

本論文では，提案システムを IA-32[18),19)] 上で構築した．本章では，提案システムを構成するモジュールと機能について述べる．**図 3** に，システムの概要を示した．実行可能ファイルが，ロードされるとき NTFS ファイルシステムを介して，メモリにロードされる．そのとき，後述する LoadImageNotifyRoutine によって，ファイルが防御対象のものであるかどうかを判定する．対象のファイルがロードされた場合は，INT01H

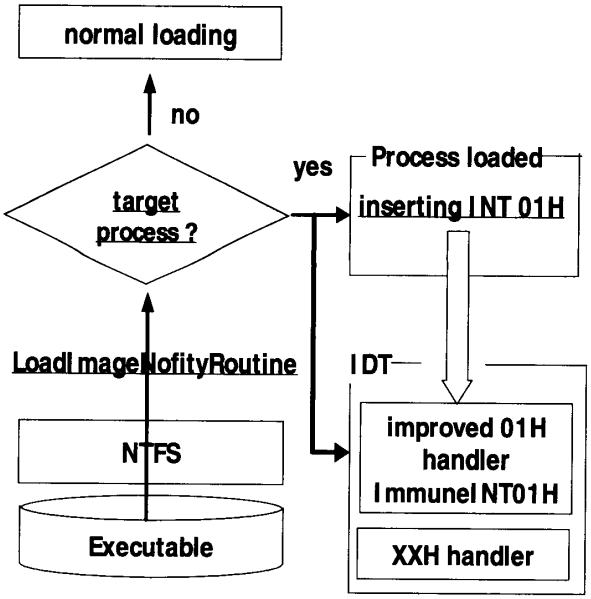


図 3 提案検出防御システム
Fig. 3 Proposal detection and prevention method.

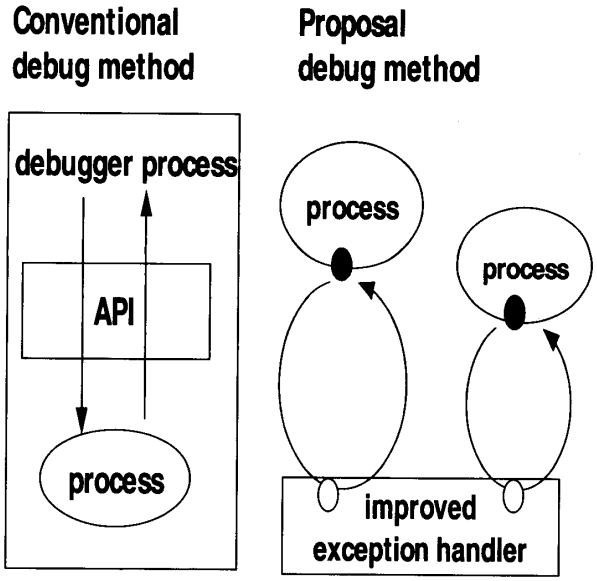


図 4 提案システムによるデバッグの自動化
Fig. 4 Automated debugging in proposal system.

を適切な位置に挿入し，提案システム用に改良された例外ハンドラを登録する．

このシステムを適用することで，実行ファイルのロード時に，次節で述べる Automated debug 属性を付与することにより，脆弱性つまりソフトウェアのバグによる感染が起こった際に，プロセスが自動的に停止あるいは対応することが可能である．個々の防御対象プロセスにおいて，Automated debug 属性を有効にするためには，後述する改良例外ハンドラと，プロセス構造ルーチンの実装が必要になる．

実行フローの観点から，提案手法は，2 段階の処理によってプロセスに実時間での自己停止を可能にする．第 1 段階として，プロセス構造ルーチンを用いて，防御対象となる実行ファイルの適切な位置に INT01H 命令を挿入する．具体的には，プロセス構造ルーチンに独自の関数を追加し，ファイルのロード時に上の処理が実行されるようにする．第 2 段階として，INT01H に対応する例外ハンドラを改良し，防御システムがフォーカスを当てている攻撃に対しての対応処理を追加する．これにより，プロセスの稼動中はつねに Automated debug 属性が付加され，有効に機能させることが可能になる．

4.1 Automated debug 属性

本節では，提案システムによってプロセスに付加する Automated debug 属性について述べる．本論文ではシグニチャベースの方式に代えて，プロセスデバッグベースの方式を提案しているが，これは，通常の自動化されていないデバッグが，デバッグ用のプロセスのメモリの中で対象となるプロセスを稼動させ，バグや欠陥がないか検査するのに対し，本論文で提示する Automated Debug では，ファイルのロード時に適切な位置にデバッグブレークポイントを設置し，例外ハンドラを改良することで，デバッグ用のプロセスを稼動させることなく，プロセスが自己のバグを検出し，自動的に停止することが可能になる．

図 4 は，上で述べた Automated debug 属性の付与されたプロセスと，通常のデバッグ方式の比較を示したものである．従来の自動化されていないデバッグでは，デバッグプロセス内にメモリを用意しないとプロセスを稼動させることができない．つまり，原則としてこのフレームワークにおいてプロセスデバッグ方式の防御方式を採用する場合，プロセスの数だけデバッグ用のプロセスを用意しなければならず，また例外ハンドラが特定の攻撃に対する防御に特化/改良されてないため，デバッグを自動化することができない．これに対し，提案手法ではプロセスに INT01H などのデバッグポイントを挿入し，例外ハンドラを改良することで，防御対象となるプロセスが，それ自体のデバッグを自動化し，感染を受けたときは自らを停止あるいは制御する属性を持つことが可能である．

具体的には，バッファオーバフローを例にとると，提案手法は，CPU の例外デバッグ/命令トレース機能を用いて，分岐命令処理をフィルタし，実行アドレスの遷移をチェックすることで不正プロセスを実時間で制御する手法である．

次節では，この Automated debug 属性を付与するための 2 段階の処理を行うために必要なプロセス構造ルーチンと，例外ハンドラの改良方法について述べる．

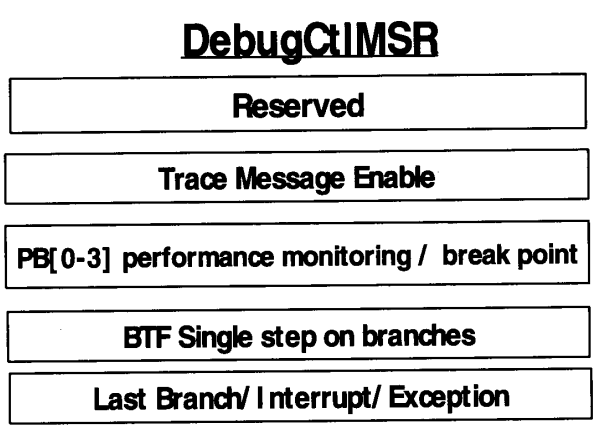


図 5 デバッグコントロール MSR のレジスタの配置
Fig. 5 Register allocation in debug control MSR.

4.2 IA のデバッグ機能

Intel ™ プロセッサでは，コードのモニタリングとデバッグのために，デバッグレジスタと 2 つのモデル固有レジスタが用意されている[20]．特に，ブレークポイント，メモリ位置，IO アロケーションのアドレスを保持するデバッグレジスタに加えて，IA-32 では，分岐，割込み，例外をモニタするための MSR（モデル固有レジスタ）が装備されているところに特長がある．

MSR は，分岐，割込み，そして例外をモニタするためのレジスタで，IA-32 では [1] DebugCtlMSR，[2] LastBranchToIP，[3] LastBranchFromIP，[4] LastExceptionToIP，[5] LastExceptionFromIP という 5 つの MSR を備えている．

図 5 は，IA-32 で用意されている，デバッグとパフォーマンスモニタ用の DebugCtlMSR のレジスタの配置を示したものである．ここで，LBR（Last Branch Register）をセットすることで，デバッグ例外が発生する直前の分岐，例外および割込み処理の IP（Instructional Pointer）のソースアドレスとターゲットアドレスを記録することが可能である．この両アドレスは，LastBranchToIP および LastBranchFromIP という 32 ビットのレジスタに格納される．本論文では，これらのレジスタをモニタし，関数が呼ばれる前後の EIP☆の遷移をチェックすることで，不正なプロセス実行の防御を行う．

4.3 プロセス構造ルーチンを用いた改良例外ハンドラの登録

Intel ™ プロセッサのマニュアルでは，同期割込みのことを例外，非同期割込みのことを割込みと定義しており，IA-32 系では数十種類の例外を発生するため，カーネルは個々の種類に対応した例外ハンドラを用意しており，提案システムでは INT01H に対応するデバッグ例外ハンドラを改良したものを実装している．これに併行して，割込みディスクリプタにおけるベクタとハンドラのアドレスの対応関係の記述も変更する．最後に，INT01H を改良し，検出に必要な情報が発生する分岐命令のみをフィルタするための ImmuneINT01H を用意する．

改良例外ハンドラの適用を，防御対象のプロセスにのみ適用するために，本論文ではプロセス構造ルーチンを用いてロード時に登録されるコールバック関数を通じて上の操作を行う．ここで，プロセス構造ルーチンとは，実行ファイルがメモリにロードされプロセスとなる際に，コールバック関数やブレークポイントを挿入するために用いるドライバ関数のことを指す．

図 3 は，コールバック関数内での操作を示したものである．提案システムでは，プロセス構造ルーチンを用いて，実行ファイルのロード時にコールバック関数を登録する．Win32 の PsSetLoadImageNotifyRoutine☆を用いると，イメージが実行時にロードされるときに呼び出されるコールバック関数を登録することができる．

```
void LoadImageNotifyRoutine{
PUNICODE_ STRING FullImageName,
HANDLE ProcessId,
PIMAGE_ INFO ImageInfo
}
```

上の関数は，提案システムでの LoadImageNotifyRoutine を示したものである．**図 6** に同コールバック関数内で行われる作業フローを示した．これについては後述するが，エントリポイントのアドレスを取得し，ハードウェアブレークポイントを挿入するというものである．具体的には，MZ ヘッダを検査し，エントリポイントに INT01H を設置する．そして，改良例外ハンドラを登録し，プロセス ID を保存する．

これにより，分岐命令実行時にはつねに改良例外ハンドラが呼び出されることになる．この改良例外ハンドラ内で，次節で説明する分岐命令処理フィルタリングが行われる．

4.4 分岐命令処理フィルタ

前節で説明したトレースコールバック関数により，提案システムでは分岐命令実行時には，つねに LastBranchFromIP が取得できるようになっている．分岐

☆ 直後に実行される命令の位置を参照する Pentium 系の CPU のレジスタの名称．16 ビットレジスタを操作するときは，E を除いて指定する．

☆ Win32 のドライバ関数作成などに用意されたプロセス構造ルーチンの 1 つ．

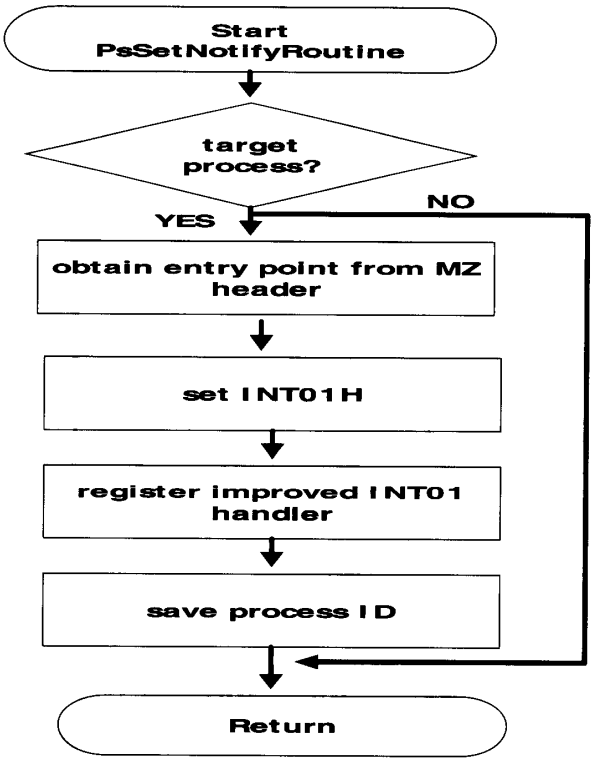


図 6 プロセス構造ルーチンを用いた INT01H と改良例外ハンドラのセット
Fig. 6 Inserting INT01H and registering improved exception handler using process structure routine.

命令処理フィルタとは，この LastBranchFromIP から，リターンアドレス改竄に利用される CALL/RET 命令をフィルタし，EIP を保存検索するためのドライバである．その意味では，本フィルタは，カーネルデバッガと似た機能を持っているが，リターンアドレスの改竄チェックに特化しているという点に特徴がある．**図 7** は，本フィルタの構造を示したものである．提案システム上で稼動するプロセスが分岐命令処理が実行されると，プロセスから INT01H が発行される．IA-32 プロセッサから LastBranchFromIP の情報が送られ BranchIP filter に制御が移る．ここで，CALL と RET 命令に応じて，BranchIP の保存と比較が行われるが，提案システムでは，スタックレコーダというメモリ構造を用意した．

カーネルデバッガは，CPU と OS の間で動作し，デバッグの対象となっているプロセスが停止すると，制御が OS からデバッガに移り，主にプロセッサのステータス情報などを取得することができる．提案システムでは，分岐命令処理フィルタを用いて OS の停止中に監視対象プロセスに対する処理を行うという，カーネルデバッガに似た機能を持っている．その結果，プロセスが不正なアクセスを受けた場合，実時間防御を行うことが可能になる．

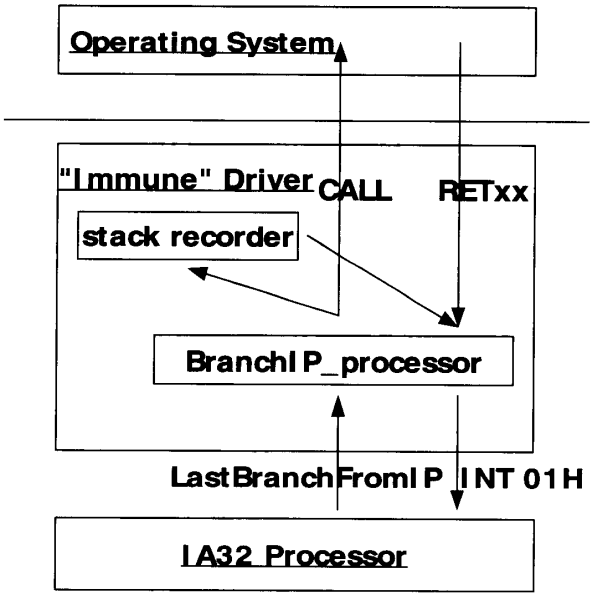


図 7 分岐命令処理フィルタ
Fig. 7 BranchIP filter.

図 8 は，本フィルタの動作手順を示したものである．提案システムでは，メイン関数のエントリポイントに，ハードウェアブレークポイントを設定しているため，分岐命令実行時にはつねに ImmuneINT01H が実行されるようになっている．結果として，監視プロセス中で実行される分岐命令について，ローカル関数呼び出しに必要な CALL と RET 命令時に LastBranchFromIP を取得でき，復帰後にアドレスの改竄が行われていないか検査することが可能になる．アドレスの改竄チェックには，スタックレコーダというメモリ構造を用意した．スタックレコーダは，分岐命令処理フィルタの中に用意した FIFO 型のメモリ構造であり，LastBranchFromIP を格納する．これにより関数の連続呼び出しが行われた場合，最後に呼び出された関数の復帰時に，実行アドレスがスタックレコーダに格納されていた場合でも，直近のレコードを削除することで対応できるという仕組みになっている．

以下に，スタックレコーダの構造体と処理方法の詳細を示す．

```
Typedef struct IMMUNE_STACK_LIST
{
LIST_ENTRY m_ListEntry;
ULONG ProcessID;
ULONG *StackPointer;
LONG CurrentStackLocation;
}IMMUNE_STACK_LIST,
PIMMUNE_STACK_LIST;
```

ここで，ProcessID とは防御対象とするプロセスの ID，StackPointer は現在の EIP の値を参照するため

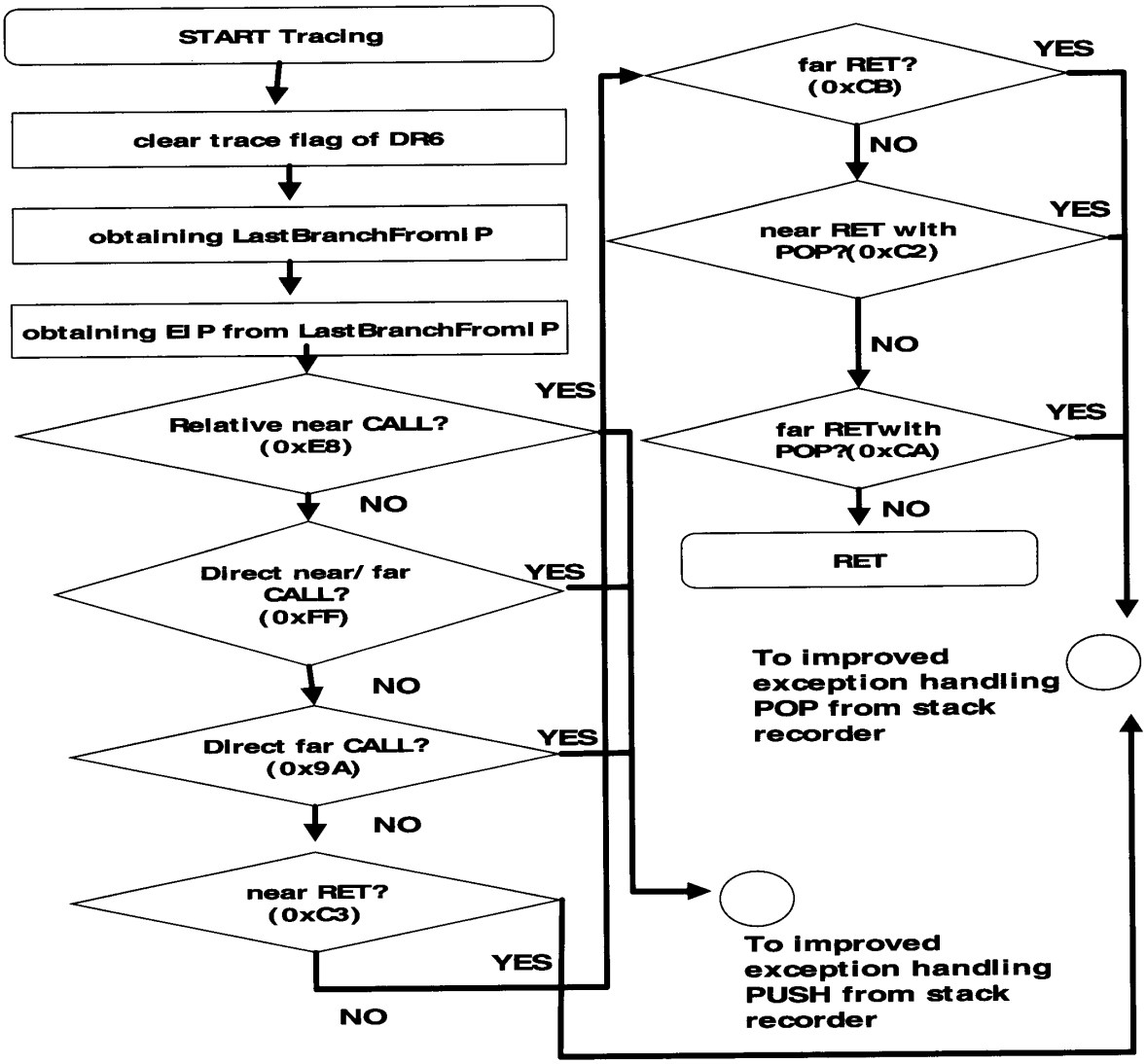


図 8 分岐命令処理フィルタの動作フロー
Fig. 8 Flow chart of BranchIP filtering.

のポインタ，CurrentStackLocation は関数が複数回呼び出された場合の，現在のスタックレコーダの長さである．分岐命令処理後に，CALL か RET かの判定をした後，CALL 命令が実行されたならばスタックレコーダに EIP の値を格納する．また，RET 命令の実行後は CurrentStackLocation の長さ分だけ，StackPointer を移動しながらスタックレコーダを検索し，現在の EIP と同じアドレスが見つかった場合は RET 処理に移り，見つからなかった場合はバッファオーバフローが起きたものとして，プロセスの停止処理を行う．

5. 検出手順

5.1 初期化

本節では，提案システムの初期化手順を示す．

手順 1 − 1：プロセスの指定
防御対象のプロセス名をレジストリから読み込む．

手順 1 − 2：コールバック関数の登録
プロセス起動時に呼び出されるコールバック関数を登録する．同関数は，プロセスのロード時に次節で述べる処理を行う．

手順 1 − 3：スタックレコーダの初期化
命令分岐処理フィルタ内で，実行アドレスの格納と検索を行うスタックレコーダを初期化する．

5.2 プロセスのロード

本節では，前節で述べたコールバック関数内での処理手順を示す．

手順 2 − 1：ターゲットプロセスの判定
手順 1 − 1 で読み込んだ名前をもとに，起動するプロセスが防御対象のものであるか判定する．

手順 2 − 2：改良例外ハンドラの登録
割込みディスクリプタを書き換えて，改良例外ハンドラを登録する．分岐命令処理フィルタは，この改良例外ハンドラ内で実行される．

手順 2 − 3：エントリポイントの取得と HW ブレー

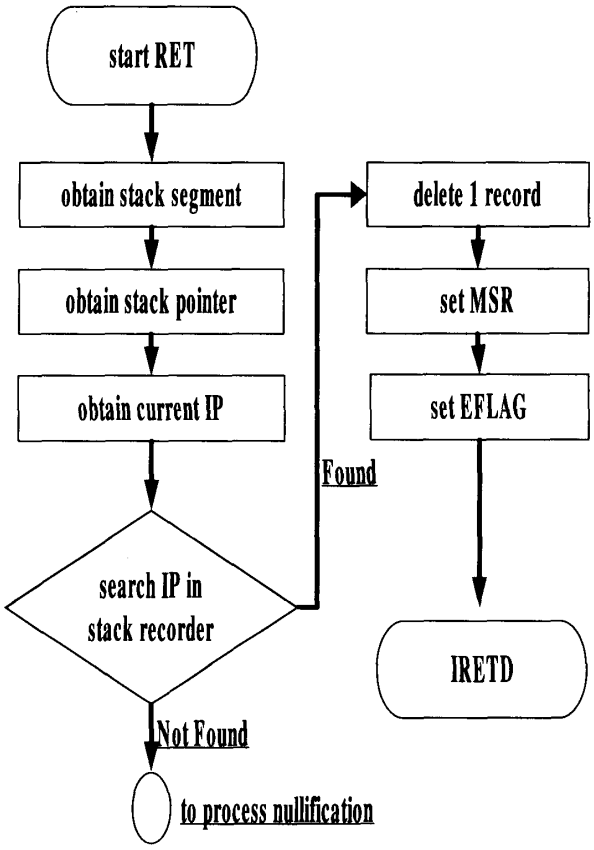


図 9 スタックレコーダを用いたリターンアドレスのチェック
Fig. 9 Checking of overwriting return address using stack recorder.

クポイントの挿入

実行プロセスの最初に実行される関数（エントリポイント）のアドレスを取得し，その直後に INT01H を呼び出すハードウェアブレークポイントを設定する．これにより，分岐命令実行時にはつねに，ImmuneINT01H がコールされることになる．

5.3 追　　跡

プロセスのロード後は，分岐命令処理フィルタによって，関数呼び出しを監視し，スタックレコーダを用いて不正な実行アドレス遷移が起きないかチェックする．

手順 3 − 1：関数呼び出し

関数呼び出しの際に，例外処理を発生させ，saved EIP（リターンアドレス）をスタックレコーダに保存する．

手順 3 − 2：関数からの復帰

図 9 に，スタックレコーダを用いたリターンアドレスのチェックの動作フローを示した．分岐命令処理フィルタによって，スタックセグメント，スタックポインタ，現在の instruction pointer の順に値が取得される．次に，現在の実行アドレスをもとに，スタックレコーダを検索する．スタックレコーダに現在の実行アドレスが格納されていれば，バッファオーバフロー発生はないとして，レコードを 1 つ削除する．検索の結果，実行アドレスが見つからなければ，saved EIP（リターンアドレス）と現在の実行アドレスが異なっており，バッファオーバフローが起こったとして，次節の処理へ進む．

5.4 プロセスの停止

提案システムの特徴は，ローカル関数実行からの復帰の際に，例外処理を発生させているため，バッファオーバフローが起こった時点で OS にプログラムコントロールを渡さずに，対応策を実施できる点にある．本論文では，バッファオーバフローが起こったプロセスが不正な処理を行う前に，当該プロセスを停止させるために，INT03H 命令を適用した．具体的には，RET 命令の次に実行される命令のアドレスを取得し，そこを INT03H 命令に書き換える．

6. 評 価 実 験

本論文では，提案システムの評価実験として，ローカルバッファオーバフローの検出時の負荷と，防御対象をデータベースサーバのプロセスに設定し，処理内容に応じてシステムの負荷を測定した．提案システムは，Intel^{TM} IA-32 の命令トレース機能を有している PentiumPro 以上のプロセッサの中から，PentiumIII 1 GHZ を選択して実装を行った．OS は，Microsoft Windows 2000 SP0 である．

6.1 リモートバッファオーバフローの検出

本節では，提案システムによるリモートバッファオーバフロー防御の評価実験について述べる．また，3 章で述べたポリモーフィックバッファオーバフローの例として，2003 年に現れた Microsoft SQL Server 2000 の脆弱性を利用する攻撃方法を検討する．

0x00b0 89e5 5168 2e64 6c6c 6865 6c33 3268 6b65
..Qh.dllhel32hke
0x00c0 726e 5168 6f75 6e74 6869 636b 4368 4765
rnQhounthickChGe
0x00d0 7454 66b9 6c6c 5168 3332 2e64 6877 7332
tTf.llQh32.dhws2
0x00e0 5f66 b965 7451 6873 6f63 6b66 b974 6f51
_f.etQhsockf.toQ

上の文字列は，オープンソースの侵入検知システムである SNORT における，2003 年 1 月 25 日 9 時時点，つまりシグニチャが SID:2003 に確定する際に，メーリングリストで議論されていたペイロードードの一部を抜粋したものである．同攻撃手法は，ポート 1434 に対してリモートバッファオーバフローを試みるため，同日約 11 時間後にシグニチャは以下のように変化した．

81 f1 03 01 04 9b xor ecx, 9B040103h
81 f1 01 01 01 01 xor ecx, 1010101h
51 push ecx
9B040103 xor 1010101 = 9A050002 = port 1434 -¿AF_INET

上の文字列は，SID:2003のシグニチャの，contentの部分を抜粋したものである．ここで，上であげた2つの文字列は，いずれも同ワームのペイロードの一部を切り取ったものであり，同種の攻撃方法に対して，複数の文字列による特徴表現が可能ありうるということが指摘される．実際に，このポート1434への送信の記述は，同ワームの特徴ではあるが，同じ攻撃方法を行うプログラムを，一般の開発環境で作成したところ，結果として機能的に等価であるが上の文字列を持たない亜種のワームが派生することが確かめられた．

以上の2形態のシグニチャはリモートバッファオーバフローを起こす同じ攻撃コードの一部を抽出したものである．提案システムでは分岐命令処理フィルタを用いた，プロセスデバッグ方式の検出を行うため，2つのシグニチャの違いにかかわらず同攻撃手法を検出し，実時間で防御を行うことが確かめられた．

表2では，上記の実験環境において，リモートバッファオーバフローのみを発生させ，提案システム稼動時と，提案システムが稼動していない場合でのシステム負荷を測定したものである．どちらのケースも防御対象プロセスにおいて，リモートバッファオーバフローが発生しており，提案システム稼動時では検出とプロセスの自動停止が行われる．結果として，提案システムが稼動しない場合と比較して負荷が高くなるが，その増分はCPU使用率1%に収まり，同実験からは，適切な負荷率で提案システムを稼動させることができるといえる．

6.2 ローカル関数呼び出しの負荷率

表3は，提案システム稼動時における，ローカル関数呼び出しの回数ごとのCPU使用率を測定したものである．この結果は，OSの構造とコンパイラの最適化の方法にも依存していると想定されるが，負荷率は関数の呼び出し回数に対して線形増加しないことが分かった．特に，関数の実行回数が1,000回以内の場合，負荷率はほぼ10%程度に収束することが判明した．また，1,000回を超えた場合でも，負荷率は線形に増加せず，10,000回のローカル関数実行時でも24%程度であるため，この点では，提案システムは適切な負荷率で稼動することが可能であることが明らかになった．

6.3 DBプロセスの防御負荷

本節で述べる評価実験では，CPU使用率について，PentiumIII 1GHZを搭載したマシンに対して，SELECTとINSERTの処理件数とデータ長さを変化させ，それぞれ50回サンプルを測定し，平均をとったものを以下の表に記載する．防御対象としたデータベースサーバは，前述したMicrosoft SQL Server 2000である．

表4は，防御対象のデータベースに行を挿入するINSERTのクエリを送信した際のシステムの負荷を測定したものである．CPUの使用率は処理件数に応じて倍になっているのが分かる．提案システムでは，1GHZのプロセッサで，1回のトランザクションに30件のクエリを処理すると50%前後の負荷がかかることが判明した．次に，監視プロセスにREAD系の処理を依頼したケースの結果を以下に示す．

表5では，防御対象のデータベースを参照するSELECTのクエリを送信した場合のシステムの負荷を測定した．READ系のクエリの場合，処理件数が増え

表2 リモートバッファオーバフロー発生時のCPU使用率（%）
Table 2 CPU utilization in preventing buffer overflow (%).

提案システム稼動	2.86
提案システム停止	2.57

表3 ローカル関数呼び出し回数とCPU使用率（%）
Table 3 CPU utilization corresponding to number of times of calling local function (%).

関数の呼び出し回数	CPU使用率
100	8.75
200	9.12
500	9.6
700	9.93
1,000	10.33
5,000	14.25
7,000	17.31
10,000	24.89

表4 DBプロセス防御時のINSERT処理時のCPU使用率（%）
Table 4 CPU utilization in DB processing INSERT query (%).

処理件数	30	10	5
提案システム稼動	47.85	17.5	9.14
提案システム停止	4.46	3.13	2.23

表5 DBプロセス防御時のSELECT処理時のCPU使用率（%）
Table 5 CPU utilization in DB processing SELECT query (%).

処理件数	30	10	5
提案システム稼動	1.48	1.21	0.97
提案システム停止	1.28	1.08	0.89

表 6 DB プロセス防御時の INSERT 処理時の CPU 使用率 (%)
Table 6 CPU utilization in DB processing SELECT query (%).

データ長 (byte)	500	200	50
提案システム稼動	1.89	1.61	1.39
提案システム停止	1.71	1.37	1.18

表 7 提案手法と他の防御手法との比較
Table 7 Comparison of Branch IP filter with previous methods.

-	リビルドの必要	制御	負荷
StackGuard/SSP	要	不可	中
Bounds Checking	要	可	大
OpenWall	不要 (カーネル以外)	不可	小
提案手法	不要	可	大

ても負荷率に影響はほとんどないといえる．ここから，データベースサーバについて，外部から READ 系のコマンドのみを受け付け，WRITE や INSERT などのコマンドは内部からのみ許可するという利用形態のケースなどに，提案システムは特に有効であるということが考えられる．

表 6 は，防御対象のデータベースプロセスに INSERT のクエリを送った際の，データの長さによって CPU の負荷率を測った結果である．実験によって，WRITE 系のコマンドを依頼する場合，処理件数によって負荷率は変化するが，データの長さによって，負荷率はほとんど変わらないことが判明した．これは，提案方式はシグニチャ方式ではなく，プロセスデバッグを機軸としているため，ペイロードをチェックせずに，一定の 32 ビット長の実行アドレスの遷移を検査するためである．提案システムがリソースについてのスケーラビリティを発揮するには，たとえば，WEB 経由のデータベースアクセスについて，WEB サーバの外部，ゲートウェイとの間に，プロキシサーバを介在させ，処理件数が多い場合，データベースサーバの CPU 使用率などに対応して，プロキシサーバに一時的にクエリをとどめておき，転送スピードを変えるなどして，大量のクエリに対して処理がドロップしないようするシステム構成などが考えられる．あるいは，通常の 3 層構造でのアプリケーションサーバの処理において，一時的にクエリを保留する方式が考えられるが，実現可能性あるいは実装面において検討すべき課題が残っている．

7. まとめと今後の課題

本論文では，CPU の例外デバッグ/命令トレース機能を用いて，分岐命令処理をフィルタし，実行アドレスの遷移をチェックすることで不正プロセスを実時間で制御する手法を提案した．提案システムは，バッファオーバフローの防御に適用され，脆弱性を孕むソフトウェアの再コンパイルを行うことなく，不正プロセスの停止が可能であることが明らかになった．**表 7** に，分岐命令処理フィルタを用いた本手法と関連研究でのべた他の手法との比較を示した．

再構築の不要な防御手法は，OpenWall と提案手法であるが，OpenWall はバッファオーバフローを検出するのみで，攻撃を受けたプロセスの停止は不可能である．一方，不正プロセスの停止が可能な手法は，Bounds Checking と提案手法である．この両者との比較において，提案手法は再コンパイルを必要としないところに特長がある．評価実験では，ローカルバッファオーバフロー検出時の負荷率と，防御対象にデータベースサーバのプロセスを設定し，クエリ処理内容に応じて実験測定を行い，適切な負荷で提案システムが稼動することが明らかになった．

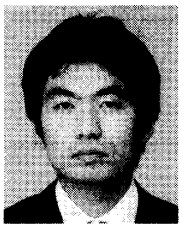
謝辞 本論文の提案システム研究は，US Air Force Office of Scientific Research (助成番号 AOARD 03-4049) の支援を受けている．また，文部科学省 21 世紀 COE プログラム次世代メディア・知的社会基盤の関係者の方からいただいた助言に謝意を記す．

参 考 文 献

1) Kim, G.H. and Spafford, E.H.: Tripwire — A File System Integrity Checker, *ACM Conference on Computer and Communications Security*, pp.18–29 (1994).
2) Roesch, M.: Snort — lightweight intrusion detection for networks, *13th Systems Administration Conference (LISA '99)*, pp.229–238 (1999).
3) Symantec Corporation: Bloodhound Technology. http://securityresponse.symantec.com/
4) 神薗雅紀，白石善明，森井昌克：仮想ネットワークを使った未知ウィルス検知システムの考察，CSS2003, No.15, pp.109–114 (2003).
5) Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q. and Hinton, H.: StackGuard — Automatic adaptive detection and prevention of buffer-overflow attacks, *7th USENIX Security Conference*, pp.63–78 (1998).
6) Etoh, H.: GCC extension for protecting applications from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp/
7) 江頭博明：Propolice スタックスマッシング攻撃検出法の改良，情報処理学会論文誌，Vol.42,

No.12 (2002).
8) Jones, R.W.M. and Kelly, P.H.J.: Backwards-compatible bounds checking for array and pointers in C programs, *AADEBUG97* (1997).
9) Ghory, Z: Openwall — Improving security with the openwall patch, *securityfocus* (2002).
10) Linux Openwall project. http://www.openwall.com/
11) Pearce, S.: Viral Polymorphism, technical paper submitted for GSEC version 1.4b (2003).
12) Szor, P. and Ferrie, P.: Hunting for Metamorphic, *Virus Bulletin Conference* (2001).
13) Takefuji, Y., Shoji, K., Miura, H., Kawade, T. and Nozaki, T.: Security strategy and a proposal of driverware, *JNSA Journal* (2003).
14) Cowan, C., Wagle, P., Pu, C., Beattie, S. and Walpole, J.: Buffer Overflows — Attacks and Defenses for the Vulnerability of the Decade, *DARPA Information Survivability Conference and Expo* (2000).
15) Pasupulati, A., Coit, J., Levitt, K., Wu, S.F., Li, S.H., Kuo, R.C. and Fan, K.P.: Buttercup — On Network-Based Detection of Polymorphic Buffer Overflow Vulnerabilities, *9th IEEE/IFIP Network Operation and Management Symposium* (2004).
16) Ando, R., Miura, H. and Takefuji, Y.: File system driver filtering against metamorphic viral coding, *WSEAS transactions of information science and applications*, Issue 4, Vol.1 (2004).
17) 独立行政法人情報処理推進機構：未知ウィルス検出技術に関する調査，15情経第1675号 (2004)
18) Intel Corporation: IA-32 Intel(R) Architecture Software Developer's Manual, Volume 2A Insruction Set Reference A-M (2004).
19) Intel Corporation: IA-32 Intel(R) Architecture Software Developer's Manual, Volume 2B Insruction Set Reference N-Z (2004).
20) Intel Corporation: IA-32 Intel(R) Architecture Software Developer's Manual, Volume 3 System Programming Guide (2004).

(平成16年11月24日受付)
(平成17年 6月 9日採録)



安藤 類央（学生会員）

平成14年慶應義塾大学大学院政策メディア研究科修士課程修了．現在，同研究科博士課程に在学中．情報通信セキュリティ，コンピュータセキュリティの研究に従事．特にIPS（Intrusion Prevention System）に興味を持つ．文部科学省21世紀COEプログラム研究員として，次世代メディア・知的社会基盤拠点形成活動に参加している．

武藤 佳恭

昭和30年生．昭和58年慶應義塾大学大学院博士課程電気工学専攻修了．工学博士．現在，ケースウェスタンリザーブ大学電気工学準教授，慶應義塾大学環境情報学部教授．情報セキュリティ，ハイパースペクトラムコンピューティング等の研究に従事．情報処理学会20周年記念論文賞（1980）．IEEE Trans. On Neural Networks功労賞（1992）受賞．米空軍研究所特別研究賞（2003年）．国際協力機構第1回JICA理事長賞（2004年）．