Vol.2012-OS-122 No.11
2012/8/1

命令レベル動的解析と関数レベル静的解析による侵入検知システムの開発

稲葉 和希[1]　白井 宏憲[1]　齋藤 彰一[1]　毛利 公一[2]　松尾 啓志[1]

概要：
攻撃から計算機を守る手段として，異常検知型の侵入検知システムが注目されている．異常検知型の侵入検知システムは，動作規則の生成方式によって動的解析方式と静的解析方式に分類できる．本稿では，命令レベル動的解析方式と関数レベル静的解析方式を組み合わせた，高検知精度の侵入検知システムを提案する．提案システムを用いることで，動的解析方式の動作規則違反数が閾値に満たない攻撃や，動作規則が未生成の領域に対する攻撃を静的解析方式によって検知でき，さらに関数ポインタを利用した攻撃を動的解析方式によって検知できる．これらより，それぞれの方式を単体で動作させる場合と比較して高精度の検知を行うことができる．

Combining Instruction-level Dynamic Analysis and Function-level Static Analysis for an Efficient IDS

Kazuki Inaba[1]　Hironori Shirai[1]　Shoichi Saito[1]　Koichi Mouri[2]　Hiroshi Matsuo[1]

Abstract: Anomaly-based intrusion detection systems (IDS) attract attention as a means to protect a computer from attacks. Anomaly-based IDS can be classified into static analysis and dynamic analysis from a viewpoint of regular execution rules (RER). In this paper, we propose a novel IDS combined instruction-level dynamic analysis and function-level static analysis. Static analysis can detect attacks to which the number of RER violations does not reach a threshold value and attacks on the domain whose RER is not generated. Further, attacks using function pointers can be detected by dynamic analysis. Consequently, combining analysises can detect more intrusions than each analysis alone.

1. はじめに

インターネットの普及とともに不正アクセスによる被害が増加しており，それらの不正アクセスの多くはプログラムのセキュリティホールに起因する．セキュリティホールは各社から提供されるパッチによって修正できるが，パッチの配布までの無防備な期間を狙って行われるゼロデイ攻撃が多発している．

ゼロデイ攻撃から計算機を守る手段として，異常検知型の侵入検知システムが注目を集めている．異常検知型の侵入検知システムは，プログラムの正常な動作を基準に検査を行うため，未知の攻撃を検知できるという特徴をもつ．異常検知型の侵入検知システムは，動作規則の生成方式によって動的解析方式と静的解析方式に分類される．動的解析方式は，監視対象のプログラムに正常な入力を与えて事前実行することで情報を収集し，その結果を基に動作規則を生成する．この方式は，実行時に定まる情報を用いた動作規則を生成できるが，一方で学習不足による誤検知が発生する可能性がある．静的解析方式はプログラムのソースコードやアセンブリを解析して動作規則を生成する．この方式は，静的に定まる情報を基に動作規則を生成するため，false positive のない動作規則を生成できる．しかし，静的ではなく実行時に定まる情報を利用した動作規則を生成できないという欠点がある．本稿では，命令レベル動的解析方式と関数レベル静的解析方式の組み合わせによる侵

[1] 名古屋工業大学
Nagoya Institute of Technology
[2] 立命館大学
Ritsumeikan University

入検知システムを提案する．動的解析方式と静的解析方式を組み合わせることによって，動的解析方式で検知できない動作規則違反数が閾値に満たない攻撃と動作規則が未生成の領域への攻撃を静的解析方式によって検知できる．また静的解析方式では検知が不可能な，関数ポインタを利用した攻撃を動的解析方式によって検知できる．これらより，それぞれの方式のみの検知を行う場合と比較して false negative を削減し，より高精度の侵入検知システムを構築できる．

2 章で関連研究について述べる．3 章で提案システムについて述べる．4 章で実装について述べ，5 章で評価について述べる．6 章で考察を述べる．7 章でまとめと今後の課題を述べる．

2. 関連研究

既存の異常検知型の侵入検知システムについて，複数の解析方式を組み合わせた既存の侵入検知システムについて述べる．さらに，本稿の提案システムの基盤となったシステムについて述べる．

2.1 異なる解析方式を組み合わせたシステム

異なる解析方式を組み合わせたシステムとして Bin らのシステム [1] や Zhen らのシステム [2] がある．Bin らのシステムは CFI[3] による命令レベルの静的解析方式を用いて，SFI[4] においてデータアクセスを行う領域の検査を行う際に不要なレジスタの退避を減らし，監視オーバーヘッドの削減を実現している．このシステムは，監視オーバーヘッドの削減をシステム統合の主な利点としている点で提案システムと異なる．また，Zhen らは，静的解析方式と動的解析方式それぞれがシステムコールの発行順を解析し，それをもとに HPDA とよばれる動作規則を生成する．このシステムは，静的解析方式と動的解析方式を組み合わせている点，さらに組み合わせる利点として広範囲な検知を挙げている点で提案システムと似ているが，Zhen らのシステムでは動的解析方式と静的解析方式それぞれがシステムコールレベルの検知を行っているのに対して，提案システムでは，動的解析部が命令レベル，静的解析部が関数レベルの検知を行っているという点で異なる．

2.2 提案システムの基盤システム

提案システムの動的解析部は，命令レベルの動的解析方式による侵入検知・セルフヒーリングシステムである RIN[5] の侵入検知部を用いた．また，提案システムの静的解析部は，関数レベルの静的解析方式による侵入検知システムである Belem を参考に実装を行った．以下，これらのシステムについて述べる．

2.2.1 動的解析方式

動的解析方式を利用した既存の侵入検知システムとして RIN がある．RIN は，命令レベルの動的解析方式により検知を行う侵入検知・セルフヒーリングシステムである．提案システムでは，動的解析部として，RIN の侵入検知部を用いた．以下に RIN の動作規則の生成と侵入検知について述べる．

RIN では，各命令の入力オペランドの不変式を動作規則として用いる．動作規則は解析部が生成する．解析部は，事前に安全な環境において監視対象プログラムを実行させて各命令の実行時の入力オペランド (レジスタやメモリ) の値を観測し記録する．その後，記録した値を解析して規則性を抽出し不変式を生成する．不変式とはプログラムの各命令において常に成立する式である．例えば，“オペランド x は定数 c1, c2, ..., cn のどれかを取る” や，“オペランド x はオペランド y よりも大きい” などである．この不変式により，オペランドの正常な範囲が分かり，値の改竄の検知や，検知された場合の修復に利用する．

侵入検知部では，アプリの状態を監視し動作規則に沿っているかどうかを確認する．事前に解析部で生成した動作規則と，実行中のアプリから得られる入力オペランドの値を，各命令の実行直前に比較する．これにより，命令によって使用されるオペランドの改竄を検知する．これはアプリケーション内で処理される値の大部分をカバーできることを意味する．しかし，動的解析方式による動作規則を使用しているため，学習の強度によりある程度の違反は発生しうる．これを侵入と誤検知するのを回避するため，発生した違反が少ない場合には異常と判断しないようにする機構が必要である．RIN では，一定異常の頻度で違反が検知された場合のみ異常と判断する．

2.2.2 静的解析方式

これまで多くの静的解析方式による侵入検知システムが提案されてきた．それらの多くは，プログラムがシステムコールを発行したときにカーネルや ptrace によってプログラムの監視を行うものである．それらの方式は，他の計算機への感染やファイルの改竄などの攻撃を行うには，システムコールの発行が不可欠であるという考えに基づく．しかし，システムコール発行時にプログラムを監視する方式では，システムコールを発行せずに終了する関数が呼びだされたか否かを確認できない．これは，コールスタックを解析して確認できる関数は，その時点で呼び出されている関数だけであり，既に終了している関数は分からないためである．そのため，プログラム実行の流れを正確に把握できず，正常な動作を偽装した攻撃を検知できない可能性が高くなる．この問題を解決したシステムとして Belem[6][7] がある．Belem では，ライブラリ関数フック用ライブラリをプログラムにリンクして，ライブラリ関数が呼び出されたときの コールスタックの確認を行う．システムコール単位で監視するより，詳細にプログラムの状態を確認できるため，より多くの攻撃を検知できる．システムコールを

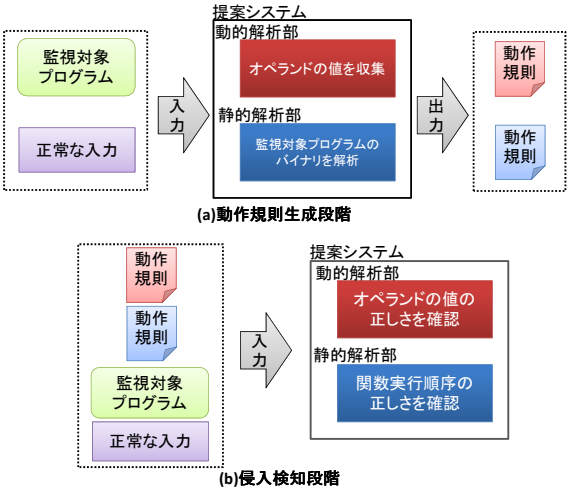

図 1 提案システムの動作図

発行せず終了するライブラリ関数でも，呼び出されたか否かを確認できる．また，システムコールがライブラリ関数から発行されているかを確認して，検知を回避して攻撃する難度を上げることができる．提案システムでは，Belem を参考とした静的解析部を実装した．

3. 提案システム

命令レベル動的解析方式と関数レベル静的解析方式を組み合わせた侵入検知システムを提案する．命令レベル動的解析方式と関数レベル静的解析方式を組み合わせることによって，動的解析部のみでは検知が不可能な動的解析部による動作規則違反数が閾値に満たない攻撃や動的解析部による動作規則が未生成の領域に対する攻撃を，静的解析部によって検知することができる．また，静的解析部のみでは検知が不可能な関数ポインタを利用した攻撃を動的解析部によって検知することができる．これらより，それぞれのシステムを単体で動作させた場合に比べて広範囲の検知が可能となる．

3.1 動作概要

提案システムの動作図を**図 1** に示す．提案システムは，命令オペランドの値についての検査を行う動的解析部と，関数の実行順序についての検査を行う静的解析部より構成される．提案システムは，動作規則の生成段階と侵入検知段階の二段階で検知処理を行う．動作規則の生成段階では，提案システムに対して監視対象プログラムと監視対象プログラムへの正しい入力を与える．動的解析部では，監視対象プログラムに対して正しい入力を与えて実行することで実行時の情報を収集して動作規則を生成する．静的解析部では，監視対象プログラムを静的解析して，関数の実行順序に関する動作規則を生成する．侵入検知段階では，動的解析部と静的解析部のそれぞれが事前に生成した動作規則を用いて監視を行いながら，監視対象プログラムを実行する．以下に両段階の処理について述べる．

表 1 異常な動作の判断

動的解析部 ＼ 静的解析部	正常と判断	異常と判断
正常と判断	正常と判断	異常と判断
異常と判断	異常と判断	異常と判断

3.1.1 動作規則の生成段階

動作規則の生成段階では，提案システムに対して，監視対象プログラムと監視対象プログラムに対する正常な入力を与える．動的解析部では監視対象プログラムに対して正常な入力を与えて実行して実行時の情報を収集し，その結果から規則性を抽出することでプログラムの各命令が満たすべき動作規則 (不変式) を生成する．静的解析部では，監視対象プログラムのバイナリを解析して関数の実行順序を抽出し，それをもとに動作規則を生成する．

3.1.2 侵入検知段階

侵入検知段階では，動的解析部と静的解析部が監視を行いながら監視対象プログラムを実行する．**表 1** に，動的解析部と静的解析部の判断に基づいて提案システムが下す判断を示す．ここで，動的解析部における異常とは，動作規則 1 つに対する違反ではなく，動的解析部における動作規則の違反数が閾値に達した状態を指す．動的解析部と静的解析部のどちらかが異常と判断した場合，提案システムは異常と判断する．これによって，静的解析方式による侵入検知システムと動的解析方式による侵入検知システムをそれぞれ単体で動作させた場合に比べて false negetive を削減でき，より広い範囲の攻撃を検知できる．

3.2 検知可能な攻撃

提案システムでは，命令レベルの動的解析方式と関数レベルの静的解析方式を組み合わせることによって，それぞれを単体で動作させる場合に比べて広範囲の検知を行うことができる．以下に，片方の解析方式のみでは検知できないが，提案システムによって検知を行うことができる攻撃について述べる．

3.2.1 規則違反が閾値に達しない攻撃

動的解析部では，プログラムにあらゆる入力をあたえ，できるだけ多くの実行フローを収集する必要がある．しかしプログラムのすべての実行フローを網羅したモデルの生成は不可能である．モデル生成時に発生しなかった実行フローがプログラム監視時に表れた場合，たとえ正常な動作であっても異常であると判断する，false positive が発生する．動的解析部が異常を検知したとき，それが false positive であるのか悪意のある攻撃によるものかを見分けることは難しい．そのため，動的解析部では，動作規則の違反数に一定の閾値を設け，動作規則との照合の際に違反数が閾値を越えた場合に異常と判断する．そのため，動的

解析部は動作規則違反が閾値に満たない攻撃を見逃す．関数のコール命令や戻りアドレスの改竄などプログラムの実行に深刻な影響を与える攻撃が行われたとしても，それらに対する動作規則違反が閾値に満たなければ検知ができない問題点がある．それに対して，静的解析部では監視対象プログラムのバイナリを静的解析して，関数の実行順序を抽出して動作規則を生成する．そのため，静的解析部による侵入検知では false positive が発生せず，動作規則に一つでも違反があれば異常と判断できる．提案システムの静的解析部では，関数の実行順序を基にした検査を行うため，動的解析部による動作規則違反が閾値に満たない攻撃でも，関数呼び出しの異常や戻りアドレスの異常を検知できる．

3.2.2 動作規則未生成部に対する攻撃

動的解析部の動作規則の生成段階において，すべての実行フローを網羅した動作規則の生成は困難である．そのため，動的解析部による動作規則には，動作規則が未生成の領域が存在する．動的解析部では，動作規則未生成領域については検査を行うことができないため，異常な関数呼び出しや戻りアドレスの書き換えが行われたとしても，異常を検知できない．それに対して，静的解析部は，監視対象プログラムのバイナリを解析して動作規則を生成するため，動的解析の動作規則が存在しない領域をふくめてすべての領域について動作規則を生成できる．提案システムの静的解析部では，関数の実行順序の検査を行うため，動的解析部による動作規則が未生成の領域でも，関数呼び出しの異常や戻りアドレスの異常については検知できる．

3.2.3 関数ポインタを利用した攻撃

関数ポインタの値は実行時に定まるため，静的解析方式によってその値の動作規則を生成できない．そのため，静的解析部では関数ポインタの値を検査できず，異常な遷移を検知できない．それに対して動的解析方式では，実行時の情報を収集して，その結果をもとに動作規則を生成する．そのため，実行時の関数ポインタの値を収集して関数ポインタの値がとるべき範囲を定めることで，関数ポインタの値について検査を行うことができる．これにより，静的解析方式のみによる侵入検知システムでは検知が不可能な攻撃を検知できる．

4. 実装

既存の動的解析方式による侵入検知・セルフヒーリングシステムである RIN の侵入検知部を提案システムの動的解析部とし，静的解析方式による侵入検知システムである Belem を参考にした静的解析部を実装して提案システムを構築した．提案システムは，動的コード変換ツール DynamoRIO[8](以下 DR) 上に実装した．以下に DR の概要と両解析部の実装について述べる．

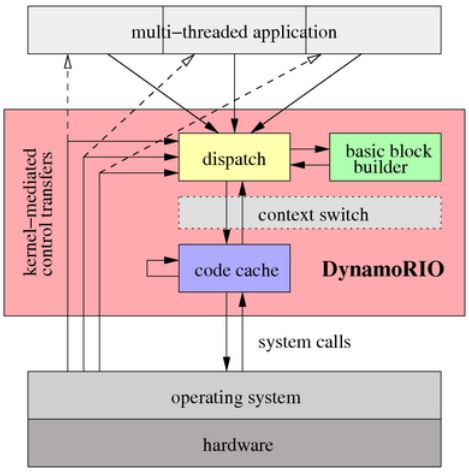

図 2 DR の動作図

4.1 DynamoRIO

DR は動的コード操作ツールの一種である．DR を用いることで，実行時に監視対象プログラムの情報を命令単位で取得したり，命令の追加や削除ができる．同様の機能のあるツールである PIN[9] や Valgrind[10] と比較して DR はパフォーマンスに優れており，プログラムの監視に適する．DR の概要図を**図 2** に示す．DR はメモリ上にロードされたマルチスレッドの監視対象プログラムを DR 内部の code cache 領域にコピーし，code cache 上で実行する．プログラムのコピーは Basic Block(以下 BB とする) と呼ばれる単位で行われる．ここで BB とは，監視対象プログラムを，コール命令，リターン命令，またはジャンプ命令で区切ったコードの一部である．そのため，BB の終端は，必ずコール命令，リターン命令またはジャンプ命令となる．code cache へのコピーは，basic block builder によって実行される命令の情報の取得や，命令の追加，削除などとの操作と一緒に行われる．この仕組みにより，監視対象プログラムの任意の場所に任意のコードを挿入できる．

4.2 動的解析部

動的解析部による検知処理は，動作規則の生成段階と侵入検知段階の二段階で構成される．動作規則の生成段階では，DR で監視対象アプリケーションを実行し，正常に実行された時の命令の入力オペランドの値を収集する．収集された値は，値の集合から規則を抽出するソフトである Daikon[11] に与えられ，動作規則が生成される．侵入検知段階では，事前に生成された動作規則と実際の動作を照合するためのコードを DR によって挿入して検知を行う．以下に動作規則の生成と侵入検知の実装について述べる．

4.2.1 動作規則生成段階

動作規則生成段階では DR の機能を利用し，全命令の入力オペランドの値を観測し記録する．これには DR の dr_insert_clean_call() 関数を呼び出す機能を使用した．この関数は，任意の場所にフックを仕掛けて指定した関数を

呼び出せる．そして呼び出された関数内では，DR 内のデータにアクセス可能であるとともに，監視対象プログラム内のメモリへも自由に参照・変更する機能が DR によって提供されている．動作規則生成段階では，この機能を使用して各命令が実行される直前の入力オペランドの値をファイルに出力する．そして記録された値の規則性を Daikon で解析し，動作規則である不変式を生成する．なお，Daikon が出力する不変式は加減算や指数，剰余などである．

4.2.2 侵入検知段階

侵入検知段階でも動作規則生成段階と同様に，DR を使用して監視対象プログラムの状態を取得する．しかし，動的解析部で dr_insert_clean_call() によって DR 関数をコールした場合，監視対象プログラムから DR と DR から監視対象プログラムへの 2 回のコンテキストスイッチが発生するため，オーバーヘッドが大きいという問題がある．そのため，提案システムの動的解析部ではアセンブリレベルの侵入検知コードを生成し監視対象プログラムに埋め込むことで検査時に DR にコンテキストが移ることなく検査を行い，実行時のオーバーヘッドを削減している．コードの埋め込みは DR の機能を利用し容易に実現できる．

4.3 静的解析部

静的解析部による処理も，動作規則の生成と侵入検知の二段階で構成される．動作規則の生成段階では，バイナリを解析して関数の実行順序に関する動作規則を生成する．侵入検知段階ではコール命令の直前とリターン命令の直前に検査用の関数を挿入して，監視対象プログラムの動作が動作規則に沿っているか検査を行う．

4.3.1 動作規則の生成

提案システムの静的解析部では，Belem と同様の動作規則を用いて検査を行う．ここで，提案システムの静的解析部で用いる動作規則について述べる．動作規則には監視対象プログラムを逆アセンブルしたアセンブリを解析し，ユーザ関数単位でユーザ関数およびライブラリ関数の呼出し順を定義する．関数 X についての正常な動作規則の生成例を**図 3** に示す．動作規則は，Entry とユーザ関数呼び出しとライブラリ関数呼び出しと Return で構成され，Entry からはじまり Return で終端する．各ノードには呼び出し元アドレスも一緒に記録する．呼び出し元アドレスによって，同一関数を複数箇所から呼び出すことがあっても区別でき，遷移先ノードを唯一つに特定できる．

4.3.2 動作規則との照合

提案システムの静的解析部は，前回のコール命令時またはリターン命令時の関数の呼び出し関係を戻りアドレスリストに記録しながら照合を行う．戻りアドレスリストは DR 内部のメモリ領域に記憶される．DR のメモリ領域はアプリケーションからアクセスできないため，戻りアドレスリストの改竄は困難である．

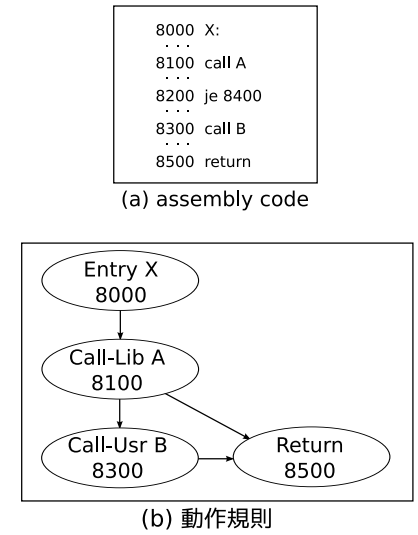

図 3 静的解析部の動作規則

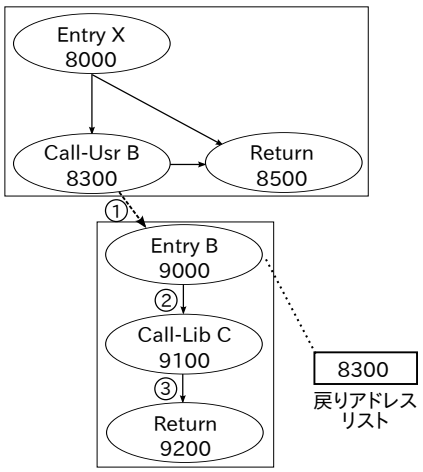

図 4 動作規則との照合例

提案システムの静的解析部では，コール命令の前またはリターン命令の前に処理を挿入できるため，コール命令とリターン命令が実行される前に必ず動作規則との照合を行うことができる．そのため，コール命令前の検査とリターン命令前の検査では，動作規則において前回検査の状態から遷移する状態の中に次に実行しようとする命令が存在するか確認を行う．検査処理について次項で詳細に述べる．

4.3.3 コール命令前の検査

コール命令前の検査では，前回のコール命令から次に実行しようとするコール命令までの遷移が動作規則に沿っているかを確認し，呼び出そうとする関数が動作規則と一致するかを確認する．まず，遷移の確認について述べる．前回の照合がコール命令前に行われたか，それともリターン命令前に行われたかを確認する．前回の照合がコール命令前だった場合には，前回のコール命令のオペランドを確認する．オペランドが関数ポインタだった場合には，遷移を確認できないため正常とし最終判断を動的解析部に任せる．

表 2　評価環境

OS	Ubuntu 10.04 32bit
kernel	2.6.32
CPU	Core 2 Duo 2.80GHz
memory	2GB

オペランドが関数のアドレスだった場合には，そのオペランドをアドレスとする関数の動作規則の先頭からの遷移先に実行しようとするコール命令が存在するかを確認し，存在しない場合は異常とする．リターン命令前だった場合には，前回のコール命令に対応する動作規則の遷移先に実行しようとするコール命令が存在するかを確認し，遷移先が存在しない場合は異常とする．遷移が正常だと判断した場合，コールする関数が動作規則と一致するかを確認し，一致すれば正常な動作とする．**図 4** を用いて動作規則との照合の例を示す．前回のコール命令のアドレスが 8300 番地で，実行しようとするコール命令のアドレスが 9100 番地の場合を考える．まず，前回の照合がコール命令のため，8300 番地をアドレスとする動作規則の遷移先を確認する．9100 番地の動作規則が存在しないため，8300 番地のオペランドである関数 UserB のアドレスの動作規則の先頭を探す (①)．そこからの遷移先の中に 9100 番地の動作規則の存在を確認できる (②)．そのため，9100 番地のコール命令は正常であると判断できる．

4.3.4　リターン命令前の検査

リターン命令前の検査では，前回のコール命令から，実行しようとするリターン命令までの遷移が動作規則に沿っているかを確認し，戻り先のアドレスが戻りアドレスリストの最も上の値と一致するかを確認する．遷移の確認では，コール命令前の処理と同様の手順で遷移の正しさを確認する．遷移の正しさを確認できたら，戻り先のアドレスが一致するかを確認する．DR では，リターン命令の第二オペランドとして戻りアドレスが設定される．この機能により，リターン命令の実行前に戻り先のアドレスを取得し，その値と戻りアドレスリストの最も上の値が一致するかを確認する．一致すれば正常な実行とする．図 4 を用いて例を示す．前回のコール命令が 9100 番地で，実行しようとするリターン命令が 9200 番地だとする．まず，9100 番地の動作規則の遷移先に 9200 番地の動作規則の存在を確認し，遷移が正しいことを確認する (③)．次にリターン命令のアドレスを取得し，戻りアドレスリストの先頭の値，この場合は 8300 番地と一致するかを確認し，一致すれば正常な実行と判断する．

5.　評価

提案システムの検知能力の検証と，監視オーバーヘッドの測定を行った．評価に使用した環境を**表 2** に示す．

```
1  int func2(void) {
2    puts("2");
3    return 0;
4  }
5  int func1(int num) {
6    void **fp = (void **)&fp;
7    puts("1");
8    func2();
9    fp[num] = (void*)func2;
10   return 0;
11 }
12 int main(int argc, char** argv) {
13   func1(atoi(argv[1]));
14   return 0;
15 }
```

図 5　攻撃プログラム 1

図 6　関数 func1 呼び出し直後のスタック

5.1　検証

侵入検知評価として，動的解析方式による動作規則違反数が閾値に満たない攻撃，動的解析方式による動作規則が未生成の領域への攻撃の侵入検知を行い，それらの攻撃を提案システムによって検知できることを確認した．

5.1.1　規則違反が閾値に達しない攻撃の検知

図 5 に示す攻撃プログラムによって検証を行った．まず，攻撃プログラムについて説明する．関数 main から関数 func1 が呼び出された直後のスタックを**図 6** に示す．図 6 は関数 main への戻りアドレス，待避済みベースポインタ，関数 func1 の引数が積まれている．関数 func1 の最初の行 (図 5 の 6 行目) `void**fp=(void**)&fp;` によって変数 fp は fp 自身，つまり図 6 のスタックの最上位のアドレスを指すことになる．そのため，fp[0] が fp 自身のアドレス，fp[2] が関数 func1 の戻りアドレスを指すことになる．fp[2] に関数 func2 のアドレスを代入して，関数 func1 の戻りアドレスを関数 func2 のアドレスに書き換えることができる．このプログラムへの引数に 2 を与えると，関数 func1 の戻りアドレスの書き換えが起こり，関数 func2 が実行される．

このプログラムは，動的解析方式による侵入検知システムでは検知できない．なぜなら，動的解析方式による侵入検知システムでは，違反数に閾値を設ける必要があるが，このプログラムの書き換え箇所は戻りアドレスのみである．そのため，動的解析方式による動作規則違反数は 1 となり閾値に達しないため，検知できない．これに対して，提案システムの静的解析部では，関数のリターン時に戻りアドレスの正しさを確認しているため，異常な関数への遷

```
1  int func1(int num) {
2      void **fp = (void **)&fp;
3      if(num)
4          return 0;
5      }else {
6          fp[2] = (void*)printf;
7          fp[4] = "num: %d\n";
8          fp[5] = (void*)num;
9          return 1;
10     }
11 }
12 int main(int argc, char** argv) {
13     if(atoi(argv[1]))
14         func1(1);
15     else
16         func1(0);
17     return 0;
18 }
```

図 7 攻撃プログラム 2

表 3 初期化オーバーヘッド (ミリ秒)

	wc	TinyHTTPd
静的解析部のみ	14	12
動的解析部のみ	204	214
提案システム	225	226

移を検知できる．

このプログラムを提案システムの動的解析部のみで実行したところ，規則違反が閾値に満たないために関数 func2 が実行されたが，提案システムをすべて動作させたところ，静的解析部が戻りアドレスの異常を検知できた．

5.1.2 動作規則未生成部の検査

図 7 に示す攻撃プログラムによって検証を行った．このプログラムは，引数として整数をとり，引数に 0 以外を与えた場合には正常に終了するが，引数に 0 を与えた場合は，関数 func1 にて戻りアドレスの書き換えが行われ，ライブラリ関数 printf が呼ばれる．ここで，動作規則の生成段階において 0 以外の値を引数に指定して動作規則を生成した．引数に 0 を与えていないので，関数 func1 において引数が 0 の場合の処理の動作規則が生成されておらず，戻りアドレスの検査を行うことができない．このプログラムに対して引数に 0 を与えたときに printf が呼び出されるかを検証した結果，動的解析部のみの監視を行った場合は異常な動作である printf が呼び出されたのに対し，提案システムでは戻りアドレスが printf となっている異常を検知できた．

5.2 監視オーバーヘッドの測定

wc コマンドと TinyHTTPd[12] を用いて，提案システムの初期化オーバーヘッドと実行監視オーバーヘッドを測定した．初期化は，動的解析部と静的解析部がそれぞれ事前に生成した動作規則を DR 上に読み込む処理である．初期化オーバーヘッドの測定結果を**表 3** に，実行監視オーバーヘッドの測定結果を**表 4** に示す．wc はそのソースプログラムである wc.c(ワード数 2358 個) を引数に実行した場合の実行時間，TinyHTTPd は 1 ギガビットのネットワークで接続されたマシン間で，apache bench[13] によってリクエスト数 100，同時接続数 5 で測定した場合の 1 リクエストあたりの平均時間を示している．ネイティブはプログラムをネイティブ実行した場合の実行時間を示し，DR のみは DR 上でコード操作を行わずに実行した場合を示している．Time が実行時間を，Ratio がネイティブの実行時間との比を表す．

表 4 実行監視オーバーヘッド (ミリ秒)

	wc		TinyHTTPd	
	Time	Ratio	Time	Ratio
ネイティブ	11	1.0	0.67	1.0
DR のみ	80	7.3	0.91	1.36
静的解析部のみ	171	15.5	1.10	1.64
動的解析部のみ	306	27.8	1.44	2.15
提案システム	329	29.8	1.56	2.33

表 5 BB 毎のオーバーヘッド (マイクロ秒)

	wc		TinyHTTPd	
	動的	静的	動的	静的
挿入数	206	206	181	181
合計実行時間	8943	1354	6952	905
平均実行時間	43	6	29	4

wc と TinyHTTPd ともに動的解析部の初期化時間が静的解析部の初期化時間と比較して非常に大きい．これは静的解析部が関数レベルの検査を行っているのに対し，動的解析部が命令レベルの検査を行っているため，動的解析部の動作規則数が静的解析部の動作規則数よりも多いためである．また，wc の実行監視のオーバーヘッドが非常に大きいのに対して，TinyHTTPd は，比較的実行監視のオーバーヘッドを抑えられていることがわかる．wc の監視オーバーヘッドが大きくなる原因として，ワード毎に文字コードなどを確認する関数の呼び出しが発生し，その都度に関数間の遷移が確認されるためと考えられる．提案システムの TinyHTTPd における実行監視オーバーヘッドは，2.33 倍であり，提案システムは，高い信頼性を求められるウェブサーバの監視等に適用可能であると考える．

5.2.1 BB 関数毎のオーバーヘッド

動的解析部と静的解析部の監視オーバーヘッドの差の原因として，BB を DR のメモリ領域にロードする際に呼ばれる関数 (以下 BB 関数とする) で行っている処理が考えられる．そこで wc と TinyHTTPd における BB 関数毎のオーバーヘッド評価を行った．評価対象は wc と TinyHTTPd であり，動作パラメータは 5.2 節の評価と同じである．その結果を**表 5** に示す．この結果より，動的解析部は静的解析部よりも約 7 倍の実行時間がかかっていることがわか

る．これは，動的解析部がほぼすべての命令について検査用コードの挿入を行っているのに対して，静的解析部では，コール命令の直前とリターン命令の直前にのみ検査用コードの挿入を行っていることによる差であると考えられる．

6. 考察

攻撃対象が提案システムに守られていると知った上でプログラムを攻撃した場合について考える．まず，DR はアプリケーションを実行する際にメモリの配置を変更するため，提案システム上でアプリケーションが実行された後にメモリ上の値の改竄を行うことは困難である．提案システムは，アセンブリ命令のオペランド値を検査する動的解析部と関数の遷移を確認する静的解析部より構成される．そのため，本提案システムに気づかれずにプログラムを攻撃するためには，関数呼び出しと戻りアドレスの改変を行わない必要がある．さらに書き換えを行う場合，動的解析部の動作規則に違反しない範囲か，または閾値に満たない範囲に抑える必要がある．そのような攻撃を行うことは困難であると考える．

7. まとめ

命令レベル動的解析方式と関数レベル静的解析方式を組み合わせた侵入検知システムを提案した．提案システムを用いることで，それぞれの解析方式を単体で動作させる場合と比較して，高精度の検査を行うことができる．今後の課題としては，オペランドの値の正しさの検査を行う，命令レベルの静的解析方式を導入することが挙げられる．命令レベル静的解析方式の導入によって，さらに検知の精度を向上させることができると考える．また，検査のレベルを動的解析方式と静的解析方式で統一して，同一の検査対象に対して同じ検査を行っている命令を削除することができ，それによりオーバーヘッドを削減できると考える．また，静的解析部によって，時刻等の変化する値を検出して動的解析部に伝えることで，動的解析部が変化する値で異常検知を出さないようにすることも考えられる．また，提案システムの動的解析部として用いた RIN は、侵入を検知した場合に値の修復を行い，アプリケーションの実行を継続するセルフヒーリング機能をもつ．現在の提案システムの静的解析部ではセルフヒーリング機能は未実装だが，将来的には実装する予定である．

参考文献

[1] Zeng, B., Tan, G. and Morrisett, G.: Combining control-flow integrity and static analysis for efficient and validated data sandboxing, *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 29–40 (2011).
[2] Liu, Z., Bridges, S. M. and Vaughn, R. B.: Combining Static Analysis and Dynamic Learning to Build Accurate Intrusion Detection Models, *Proceedings of the Third IEEE International Workshop on Information Assurance*, pp. 164–177 (2005).
[3] Abadi, M., Budiu, M., Erlingsson, U. and Ligatti, J.: Control-Flow Integrity Principles, Implementations, and Applications, *ACM Conference on Computer and Communication Security*, pp. 340–353 (2005).
[4] Wahbe, R., Lucco, S., Anderson, T. E. and Graham, S. L.: Efficient software-based fault isolation, *SOSP '93 Proceedings of the fourteenth ACM symposium on Operating systems principles*, pp. 203–216 (1993).
[5] Shirai, H., Saito, S., Mouri, K. and Matsuo, H.: Monitoring Instruction-based Intrusion Detection and Self-Healing System, *Proceedings of the International Multi-Conference of Engineers and Computer Scientists 2012*, Vol. I, pp. 250–256 (2012).
[6] Kato, Y., Makimoto, Y., Shirai, H., Shimizu, H., Furuya, Y., Saito, S. and Matsuo, H.: Monitoring Library Function-based Intrusion Prevention System with Continuing Execution Mechanism, *Proceedings of the 2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, pp. 548–554 (2010).
[7] 槙本裕司，齋藤彰一，古屋雄介，白井宏憲，上原哲太郎，松尾啓志：ライブラリ関数呼び出し監視による侵入防止システムの実現，情報処理学会論文誌コンピューティングシステム，Vol. 3, No. 1，pp. 38–49 (2010).
[8] Bruening, D.: Efficient, transparent, and comprehensive runtime code manipulation, PhD Thesis, Massachusetts Institute of Technology (2004).
[9] Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. and Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation, *ACM SIGPLAN Notices*, Vol. 40, No. 6, pp. 190–200 (2005).
[10] Nethercote, N. and Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation, *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pp. 89–100 (2007).
[11] Ernst, M., Perkins, J., Guo, P., McCamant, S., Pacheco, C., Tschantz, M. and Xiao, C.: The Daikon system for dynamic detection of likely invariants, *Science of Computer Programming*, Vol. 69, No. 1–3, pp. 35–45 (2007).
[12] Tiny HTTPd's tiny homepage, http://tinyhttpd.sourceforge.net/.
[13] ApacheBench, http://httpd.apache.org/docs/2.0/programs/ab.html.