社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2009－OS－110 (2)
2009／1／28

システムコールとライブラリ関数の監視による侵入防止システムの実現

槙本裕司[†1] 鶴田浩史[†1] 齋藤彰一[†1]
上原哲太郎[†2] 松尾啓志[†1]

ゼロデイ攻撃や未知の攻撃から計算機を守る手法として侵入防止システムの研究が多く行われている．既存の侵入防止システムはシステムコール呼び出し時に得られる情報のみに基づいているため，実行時の状態遷移の把握が十分ではなく，true negative あるいは正常な実行であるかのように偽装した攻撃による耐性が十分であるか疑わしい．本研究では，ライブラリ関数呼び出しごとにコールスタックを調べることで，プログラムの実行状態を詳細に把握する手法を提案する．また，ライブラリ関数の呼び出しと終了を把握することで，ライブラリ関数が呼び出されている時のみシステムコールの発行を許可する．提案システムを Linux 上に実装し評価を行った．

Implementation of a Intrusion Prevention System Based on Monitoring System Call and Library Function Call

Yuji Makimoto,[†1] Koji Tsuruta,[†1] Shoichi Saito,[†1]
Uehara Tetsutaro[†2] and Hiroshi Matsuo [†1]

Some intrusion prevention system has been studied to prevent a zero-day attack and an unknown attack. The existing systems don't hold enough program execution statuses; because the systems use only system call's past record. In this paper, we propose a novel intrusion prevention system, named Belem, by monitoring both a system call and a library function call. Belem checks a call stack before a library function is executing. We implemented Belem on Linux, and evaluated it.

1. はじめに

セキュリティホールが見付かってから，パッチが適応されるまでの無防備な期間をねらって行われるゼロデイ攻撃が増加している．ゼロデイ攻撃から計算機を守る手段として，異常検知方式を用いた侵入防止システムが注目されている．異常検知方式ではあらかじめプログラムの正常な動作規則を作成しておき，プログラムを正常な動作規則と照合しながら実行することによって，異常な関数の呼び出しやシステムコールの発行を見つける方式である．この方式は，プログラムのセキュリティホールをあらかじめ把握していなくても，攻撃によるプログラムの異常な動作を検知可能である．そのため，ゼロデイ攻撃に加えて未知の攻撃も防ぐことができる．

悪意のあるユーザから攻撃を受けたプログラムは，通常の実行では現れない関数の呼び出しやシステムコール発行が行われる．つまり攻撃を受けたプログラムは，自身の実行ファイルとは異なった動作をすることになる．そのため多くの異常検知方式の研究では，プログラムが正常に動作していたときの情報以外に，プログラムのソースコードや実行ファイルから正常な動作規則を作成する．そして，プログラム実行の監視は，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 によって行う手法が多い [1~4]．この手法は，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が不可欠である，という考えに基づいてる．つまり，システムコールを発行せずに有効な攻撃ができないこと，また，システムコールのフックはカーネルや ptrace において容易に行えること，さらに，カーネルや ptrace によってコールスタックの確認を行うことが可能であるなど，システムコール発行時はプロセスの監視に適しているためである．

しかし，システムコール発行時にプログラムを監視する手法では，システムコールを発行することなく終了する関数が呼び出されたか否かを確認することはできない．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することができず，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．

この問題を解決するために本稿では，ライブラリ関数フック用ライブラリ libelem(lib effective library executing monitor) をプログラムにリンクすることで，ラ

†1 名古屋工業大学
Nagoya Institute of Technology
†2 京都大学
Kyoto University

イブラリ関数が呼び出されたときのコールスタックを確認する手法を提案する．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，検知できる攻撃が多くなる．システムコールを発行することなく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することができる．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することで，検知を回避して攻撃することの難度を上げることができる．

本論文では 2 章で関連研究について述べる．3 章で我々の提案手法について述べる．4 章で実装について述べ，5 章で考察を述べる．6 章でまとめと今後の課題を述べる．

2. 関連研究

関連研究での正常な動作規則の作成法と，プログラムの実行監視法について述べる．

2.1 正常な動作の定義作成法

プログラムを実際に実行させて収集した情報を基に，正常な動作モデルを作成する手法[1)2)]がある．この手法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を作成することは非常に困難である．モデル作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誤検知する．この誤検知を false positive と呼ぶ．プログラムを監視しているシステム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．

一方，実行ファイルあるいはソースコードを静的解析することにより正常な動作規則を作成する手法[3)4)]もある．この手法では，プログラムの実行フローをすべて網羅した動作規則を作成することができる．そのため，プログラム監視時に false positive が発生しない動作規則を作成することができる．

これらの手法では関数単位で関数の呼び出し順を定義した動作規則を作成することで，n 番目のシステムコールを発行したときの状態から，n+1 番目のシステムコールを発行したときの状態に至るまでの，関数の呼び出し及び終了の正当制を確認できる．

2.2 プログラムの実行監視法

プログラムの実行監視手法はシステムコール発行時に行うものと，ライブラリ関数呼出し時に行うものとに分けられる．

2.2.1 システムコール発行監視手法

監視対象プログラムの実行監視をシステムコール発行時に行う手法[2〜4)]は多い．Wagner らの手法[3)]ではプログラムが発行したシステムコールのみを用いて動作規則との照合を行っているため，遷移先の状態を特定することが困難である．一方，Feng らの手法[2)]や安部らの手法[4)]では，監視対象のプログラムがシステムコールを発行したときの，コールスタックを用いることで遷移先の状態を特定しやすくしている．

状態を特定するためにコールスタックを用いる手法の有用性について述べる．C 言語で記述されたプログラムは通常，関数が呼び出されるごとにフレームがコールスタックに積まれる．フレームには関数の戻りアドレスや呼び出し元関数のベースポインタの値が格納される．そのため，コールスタックに積まれているベースポインタの値を利用することで効率よく関数の戻りアドレスを収集することができる．また，戻りアドレスから呼び出されている関数を知ることができる．さらに，システムコールが前回発行されたときからのコールスタックの変化を調べることで，どの関数が終了し，どの関数が新たに呼び出されたのかがわかる．しかし，システムコール発行時のコールスタックを用いて確認できる関数は，システムコールが発行されたときに実行途中の関数だけである．つまり，システムコールを発行することなく終了してしまった関数が呼び出されていたことを確認することはできない．

安部らの手法[4)]を例に述べる．安部らの手法で用いている正常な動作規則は，関数単位で関数の呼び出し順を定義したものである．関数 X についての規則が図 1 のように表されていたときを考える．図 1 は，関数 X が A→B→C または A→D の順に他の関数を呼び出した後，関数 X が終了するということを表している．監視対象のプログラムで n 回目のシステムコールが発行されたときには関数 X から関数 A が呼び出されており，n+1 回目のシステムコールが発行されたときには関数 X から関数 B が呼び出されていたとすると，関数 A が終了した後関数 B が呼び出されたと考えることができ，正常な動作規則とも一致する．

安部らの手法では，関数 B が特定の条件下でのみシステムコールを発行する関数であった場合，以下のような特別な措置をとる必要がある．関数 B からシステムコールが発行されなければ，プログラム監視時に関数 B が呼び出されたか否かを確認することができない．そのため，n+1 番目のシステムコールが発行されたときに関数 C が呼び出されていたなら，正常な動作規則においては関数 B はシステムコールを発行せずに終了したと判断する．これは正常な動作モデルに，関数 A が終了した後に関数 C が呼び出されるという遷移 (図 2 の遷移 ①) がはじめから存在していたのと同意である．つまり，静的解析で作成された動作規則はプログラムの詳細な状態遷移を表わすことができるが，監視しているプログラムの状態の把握をそれと同程度に詳細に行えないため，許容される遷移が多くなる．

ここで，関数 B を用いた攻撃を考える．関数 X が関数 B を呼び出しているときに，攻撃者が関数 B の戻りアドレスを書き換えて，関数 B の次に関数 D を呼び出させる攻撃がある．このとき関数 B がシステムコールを呼び出したら，監視時に把握できる関数の呼び出し順は A→B→D(図 2 の遷移 ②) となり，動作規則に定義されていない呼び出し順であるため検知できる．しかし，関数 B がシステムコールを呼び出さずに終了した場合，

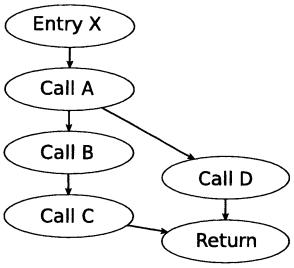

図 1 関連研究の正常な動作規則例

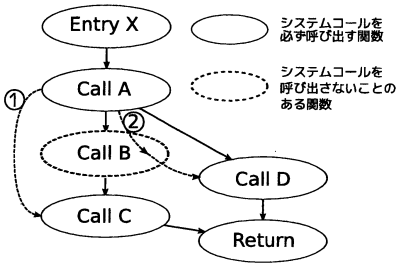

図 2 関連研究の正常な動作規則の問題点

監視時に把握できる関数の呼び出し順は A→D となり，動作規則と一致する．よって異常検知失敗となる．

2.2.2 ライブラリ関数実行監視手法

プログラムの実行監視にシステムコール発行時以外の情報を用いる手法について述べる．Gaurav らの提案する e-NeXSh[5] では，攻撃者にとって有用なライブラリ関数とそこから呼び出されるシステムコールに対して特別な処理を行う．まず，ライブラリ関数をフックし，コールスタックの正当性を確認する．そして，ライブラリ関数の呼び出しが正常に行われたことを証明するためのシステムコールを発行する．このシステムコールにより，e-NeXSh のカーネルはライブラリ関数が正常に呼び出されたことを確認した上でライブラリ関数から発行されたシステムコールの処理を行う．これにより，攻撃者は不正な手順で有用なライブラリ関数あるいはシステムコールを呼び出すことができない．また，e-NeXSh ではコールスタックの確認も行い，戻りアドレスが正当な場所を指しているかを確認することで，ライブラリ関数の呼び出し元の正当性の確認も行う．e-NeXSh によるライブラリ関数のフックオーバヘッドは大きいが，フックするライブラリ関数を攻撃者にとって有用なものに限定しているため，監視時のオーバヘッドはそれほど大きくない．

3. 提案手法

本稿では以下の 3 つの手法を提案する．

(1) システムコールを発行する可能性のあるライブラリ関数すべてをフックする
(2) システムコール処理前にライブラリ関数がフックされていることを確認する
(3) フックして得られたコールスタックの情報をユーザ空間に保持する

これら 3 つの提案および動作規則の構成について詳しく述べる．

3.1 ライブラリ関数のフック

システムコール発行時の情報だけではシステムコールを呼び出さずに終了した関数の情報を得られない．そこで，システムコールを発行する可能性のあるライブラリ関数をすべてフックする．ライブラリ関数呼び出し時点の情報を用いることで，システムコールを発行しなかったライブラリ関数の情報も得て，きめ細かにプログラムの実行を把握する．これによって 2.2.1 で述べた異常な遷移を検知できる．

また，システムコールを呼び出さずに有効な攻撃を行うことは困難であるため，システムコールを呼び出さない関数の危険度は低いと考える．そのため，ライブラリ関数ごとにシステムコールを呼び出す可能性があるか否かをあらかじめ静的解析により調べておき，システムコールを呼び出す可能性のあるライブラリ関数のみフック対象とする．フックするライブラリ関数を減らすことでオーバヘッドの増加を抑えることができる．

3.2 ライブラリ関数のフック確認

フック対象のライブラリ関数の呼出し時にフラグを立て，終了時にフラグを下げることで，フック対象のライブラリ関数がフックされて実行中であるか否かを判断することができる．システムコールは必ずフック対象のライブラリ関数から発行されるため，フラグが立っているときのみシステムコールの発行を許可する．これによって，libelem を回避してシステムコールを呼び出す攻撃を検知することができる．

3.3 フック時の処理の高速化

e-NeXSh では，ライブラリ関数をフックした後，ライブラリ関数が正常に呼び出されたことをシステムコールによってカーネルに知らせる．しかし提案手法ではそのシステムコールを省き，ユーザ空間のメモリに記憶しておくことでオーバヘッドの増加を抑える．このメモリをコールスタック履歴と呼ぶ．

ライブラリ関数をフックしたときのコールスタックの情報をこのコールスタック履歴に書き込む．そして，カーネルがプログラムの動作の確認をする度に，コールスタック履歴にアクセスする．

3.4 動作規則の構成

提案手法で用いる正常な動作規則について述べる．

3.4.1 動作規則の作成単位

動作規則はユーザ関数単位で，ユーザ関数及びライブラリ関数の呼び出し順を定義する．本稿の動作規則は，安部らの手法で用いているものと似ているが，我々の手法では関数の呼び出し元のアドレスも用いている点で異なる．それによって，単一の関数から同じ関数が複数回呼び出されていたとしても，それらを区別することができる．

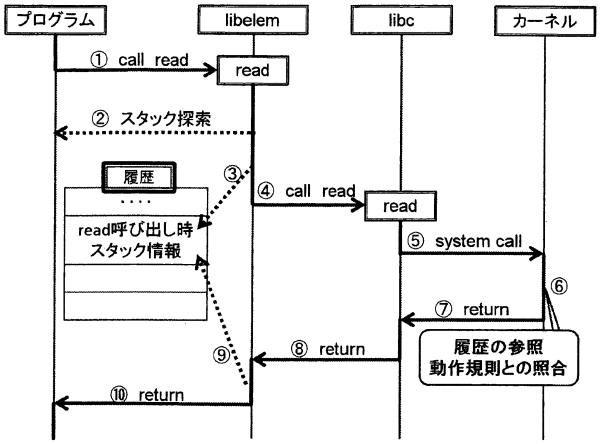

図 3 提案システムの全体像

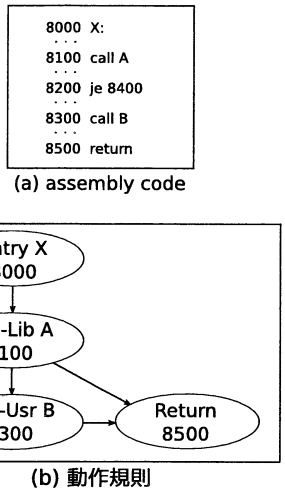

図 4 動作規則の作成

3.4.2 動作規則に定義する関数

動作規則に定義するライブラリ関数は，本提案手法でフックを行っているものに限定する．そのため，動作規則で定義されているライブラリ関数は，監視時にも必ず確認がとれる．また，監視時に確認できたライブラリ関数は必ず動作規則に定義されている．そのため，2.2.1 で述べたような，ライブラリ関数呼び出しが確認できない場合がなくなり，遷移の増加を抑えることができる．

3.4.3 動作規則の作成方法

false positive が起きないようにするために，静的解析により動作規則を作成する．監視対象の実行ファイルを逆アセンブルしたものを解析するプログラムにより，動作規則を出力する．そのため，動作規則に関する知識がないユーザでも容易に作成できる．

4. 実　　装

提案システムを Linux 上に実装した．ライブラリ関数のフックはライブラリ libelem を監視対象プログラムに優先的にリンクすることで実現した．システムコールのフックは Linux カーネルを書き換えることで実現した．監視対象は C 言語で記述されたプログラムである．

4.1 全体構成

提案システムの全体構成を図 3 を用いて述べる．図 3 は，監視対象プログラムが read 関数を呼び出したときの処理の流れを示している．監視対象プログラムが read 関数を呼び出すと，最初にリンクされている libelem に定義されている read 関数が実行される (①)．libelem の read 関数ではコールスタックを探索し，read 関数を呼び出しているユーザ関数を調べる (②)．その結果をコールスタック履歴に追記する (③)．その後，libc の read 関数を呼び出す (④)．libc の read 関数がシステムコールを呼び出すと (⑤)，カーネルがコールスタック履歴と動作規則の照合を行う (⑥)．システムコールが終了し (⑦)，libc の read 関数も終了すると (⑧)，libelem の read 関数に戻る．そこで libc の read 関数が終了したことをコールスタック履歴に記録する (⑨)．最後に libelem の read 関数が終了し，プログラムに戻る (⑩)．

4.2 動作規則の作成

動作規則には監視対象の実行ファイルを逆アセンブルしたものを解析し，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び出し順を定義する．関数 X についての正常な動作規則の作成例を図 4 に示す．Entry，Return，ユーザ関数呼び出し，ライブラリ関数呼び出しを表すノードで構成され，Entry からはじまり Return で終端する．call 先の関数がユーザ関数であるかライブラリ関数であるかは，call 先のアドレス範囲を確認することで区別する．解析時にジャンプ命令を考慮してノード間の遷移先を決める．各ノードには呼び出し元アドレスもいっしょに記録する．呼び出し元アドレスがあることによって，単一の関数から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することができる．動作規則の先頭にはインデックスを付け，各ユーザ関数の動作規則の Entry ノードを効率よく探すことができる．

作成した動作規則のサイズを表 1 に示す．表の実行ファイルとは動作規則の作成元のファイルである．text セクションは実行ファイルの中に含まれているセクションで，ユーザが作成した実行コードがここに含まれる．動作規則は実行ファイルの text セクションを解析することで作成する．wc と inetd の動作規則はどちらも text セクションの約 1.5 倍となっている．inetd においては動作規則が実行ファイルの 60%と大きい．今後，動作規則のサイズを小さくできるようにフォーマットの改良を行う．

表 1　サイズの比較

プログラム名	実行ファイル [KB]	text セクション [KB]	動作規則 [KB]
wc	85	9	13
inetd	30	11	18

4.3　コールスタック履歴の作成

環境変数 LD_PRELOAD を利用することで，監視対象のプログラムが呼び出す libc 関数をフックした．フックする必要がある libc 関数と同じ型の関数を定義したライブラリ libelem を作成し，環境変数 LD_PRELOAD に libelem のアドレスを指定した．LD_PRELOAD に指定されているライブラリは libc より先にリンクされるため，libc の関数を呼び出す代わりに libelem の同型の関数を呼び出させることができる．

libc 関数をフックした後，呼び出されているユーザ関数のアドレスをコールスタックから調べ，libelem で確保したメモリ領域であるコールスタック履歴に追記する．コールスタック履歴は，呼び出そうとしている libc 関数のアドレス，呼び出されている関数の戻りアドレス，libc 関数を実行中であるか否かを表すフラグの 3 つで構成される．コールスタック履歴の情報は次にシステムコールが呼び出されて，動作規則との照合が行われるまで保持される．

フックした libc 関数がシステムコールを呼び出さなかったとしても，libc 関数を呼び出した記録がコールスタック履歴に残る．また，libc 関数がユーザ関数にリターンする前にも libelem の関数を通るため，フックしている libc 関数が終了したことをコールスタック履歴に記録する．これにより，libc 関数を実行中か否かの確認を行うことができる．これを利用して，libc 関数を呼び出していないときにはシステムコール発行を禁止することができる．例えば，攻撃者がスタックを正常であるかのように偽装したとしても，libelem から libc 関数を呼び出していなければ異常であると判断できる．

システムコールフック機構から参照されるまでの記録を保持する必要があるため，コールスタック履歴には十分なサイズのメモリを割り当てる．

4.4　動作規則との照合

監視対象のプログラムのシステムコールをフックし，コールスタック履歴を参照して動作規則との照合を行う．異常な動作であると判断したときは，本来処理する予定のシステムコールの代わりに kill システムコールを呼び出し，監視対象のプログラムの実行を停止する．攻撃者の意図したシステムコールを処理することなくプログラム自体を停止することができる．

動作規則との照合は，まずシステムコールが前回呼び出されたときの実行状態に一致する動作規則上のノード，および戻りアドレスリストを記憶しておき，そこから次に遷移できるノードを調べる．そして，次に呼び出されたライブラリ関数までの遷移を確認する．戻りアドレスリストは動作規則との照合を行う過程で，辿ったユーザ関数の戻りアドレスを記憶しておくためのリストである．

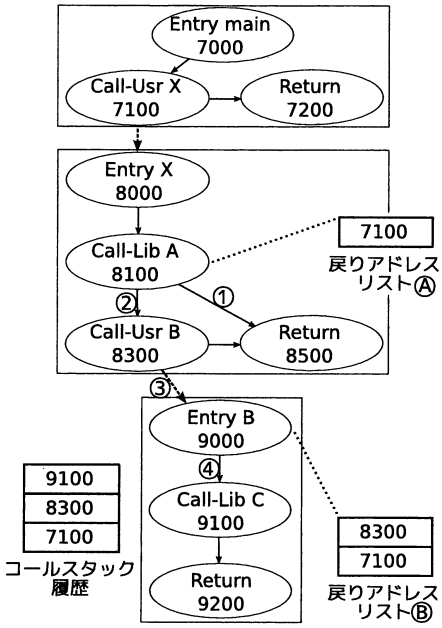

図 5　動作規則との照合

図 5 を用いて動作規則との照合の方法を述べる．システムコールが前回呼び出されたときの実行状態が，図 5 の 8100 番地からのライブラリ関数 A の呼び出し (Call-Lib A) であったとする．次に遷移するノードが Return ノードであれば (①)，戻りアドレスリスト Ⓐ の一番上に積まれているアドレスのノード (関数 X の呼び出し元) に遷移する．また，次に遷移するノードがユーザ関数呼び出しノードであれば (②)，ユーザ関数の呼び出し元となる自身の実行アドレスを戻りアドレスリスト Ⓐ に積んだ後，そのユーザ関数に対応した動作規則の Entry ノードを探し (③)，さらにそこから遷移できるノードを調べる．次に遷移するノードがライブラリ関数呼び出しであれば (④)，この実行位置がコールスタック履歴と一致するかを確認する．コールスタック履歴に書かれている戻りアドレスと，動作規則に書かれている呼出元アドレスおよび戻りアドレスリスト Ⓑ に積まれているアドレスとの対応を調べる．これらのアドレスが対応していれば，正常な動作であると判断して動作規則との照合を終了し，対応していなければ他の遷移可能ノードについて調べる．ライブラリ関数呼び出しノードに到達できなければ異常と判断する．

監視しているプログラムに対してシグナルが送られたとき，シグナルハンドラが登録されていたら，そのシグナルハンドラのアドレスを取得する．シグナルハンドラも他のユーザ関数と同様に動作規則を作成するので，シグナルハンドラに対応した動作規則の Entry ノードから照合する．

監視対象プログラムが execve システムコールを呼び

表 2　監視オーバヘッド

プログラム名	通常実行	ライブラリ関数フック	動作規則と照合
wc	1.39(1.00)	1.86(1.34)	2.39(1.72)
inetd	2.53(1.00)	2.60(1.03)	3.21(1.27)
inetd(local)	1.03(1.00)	1.11(1.08)	1.50(1.49)

単位は秒，括弧内は倍率

出したときには一度監視を打ち切り，新たに実行されるプログラムが動作規則を持った所定の形式であれば，監視が開始される．また，監視対象プログラムが fork, clone システムコールを呼び出したとき，子プロセスも監視対象とする．

5. 評価と考察

前章で実装したシステムの評価と考察を行った．評価には，監視オーバヘッドと branching factor を用いた．

5.1 監視オーバヘッドの測定

wc 及び inetd を監視したときのオーバヘッドを表 2 に示す．wc は 15MB のファイルを引数とした場合の測定結果で，inetd は別ホストから 1000 回接続を行った場合，inetd(loacl) は同じホストから 1000 回接続を行った場合の測定結果である．測定を行ったのは，通常通りプログラムを実行した場合，libelem を用いてライブラリ関数のフックのみを行った場合，提案システムにより動作規則と照合を行った場合である．このとき wc はシステムコールを約 1000 回フックし，ライブラリ関数を約 1500 万回フックした．inetd はシステムコールおよびライブラリ関数をどちらも約 100 万回フックした．

wc は他に比べて libelem を用いてライブラリ関数のフックのみを行ったときのオーバヘッドが大きい，これはフックしたライブラリ関数がシステムコールを発行しなかった回数が非常に多かったためである．次に，inetd は別ホストから接続を行った方の倍率が小さいが，ライブラリ関数をフックしたときのオーバヘッドは別ホストで 0.07 秒，同一ホストで 0.08 秒でありほとんど同じであった．また，動作規則と照合時のオーバヘッドも別ホストで 0.68 秒，同一ホストで 0.47 秒となっており近い値であることがわかる．このことから，オーバヘッドはフックしたライブラリ関数の数及びフックしたシステムコールの数に大きく依存していることがわかる．

5.2 branching factor による評価

動作規則の検知精度の評価には branching factor[3)6)] を用いた．branching factor は動作規則に含まれている遷移の総数をノードの総数で割った値である．branching factor が小さいほど，次に遷移可能なノードが限定されるため，プログラムの実行を厳しく制限することができ，検知を回避して攻撃を行うことが困難になる．

提案手法により，ライブラリ関数の呼び出しが必ず把握できるようになったことで，どれだけ検知精度が改善されたかを調べた．比較対象としてライブラリ関数の呼び出しが把握できるとは限らない場合を考慮した動作規則の branching factor の値を用いた．

表 3　ライブラリ関数呼び出しが必ず把握できるとき (提案手法) の branchig factor

プログラム名	branching factor	遷移の総数	ノードの総数
wc	1.33	312	234
inetd	1.48	574	389

表 4　ライブラリ呼び出しが把握できるとは限らないときの branchig factor

プログラム名	branching factor	遷移の総数	ノードの総数
wc	1.57	367	234
inetd	2.01	783	389

ライブラリ関数の呼び出しが把握できるとは限らない場合の動作規則の作成法について述べる．まず，システムコールを呼び出さない可能性のあるライブラリ関数を特定する．そして，これらの関数の呼び出しが確認できなかったときのための遷移を動作規則に追加する．提案手法での動作規則が図 6 のように表されており，関数 B はシステムコールを呼び出さない可能性のあるライブラリ関数であった場合について考える．ライブラリ関数の呼び出しが把握できるとは限らない場合，関数 B がシステムコールを呼び出さずに終了すると，関数 B が呼び出されたか否かを確認できない．関数 B の呼び出しが確認できなかったとき，関数 B の直前のノードから，関数 B から遷移できるノードに直接遷移することになるので，動作規則は図 7 のようになる．

ライブラリ関数の呼び出しが必ず把握できる場合の branching factor を表 3 に，把握できるとは限らない場合の branching factor を表 4 に示す．システムコールを呼び出さない可能性のあるライブラリ関数を呼び出すノードは，wc で 23 個，inetd で 76 個であった．システムコールを呼び出さない可能性のあるライブラリ関数呼び出しノードが多い inetd の方が提案手法により branching factor が大きく改善されている．

5.3 検知可能な攻撃

攻撃者がスタックに攻撃コードを挿入し，関数の戻りアドレスをスタック上のコードに書き換えることで，攻撃コードを実行させる攻撃について考える．この攻撃によりスタック構造が壊れ，スタックに積まれている ebp を辿って，main 関数までの戻りアドレスを取得できなければ検知できる．また，攻撃者が正常に動作しているようにスタックを偽装した上で，システムコールを発行した場合でも，コールスタック履歴にライブラリ関数が実行中であるという記録が残らないので検知できる．

攻撃者が関数の戻りアドレスを書き換え，ライブラリ関数に制御を移す攻撃について考える．不正に呼び出されたライブラリ関数からシステムコールを発行したとき，コールスタック履歴にはライブラリ関数が実行中であるという記録が残っていないので検知できる．また，不正に呼び出されたライブラリ関数からシステムコールが発行されなかった場合でも，コールスタック履歴に残って

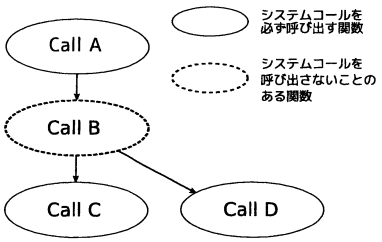

図 6 ライブラリ関数呼び出しが必ず把握できるとき (提案手法) の動作規則

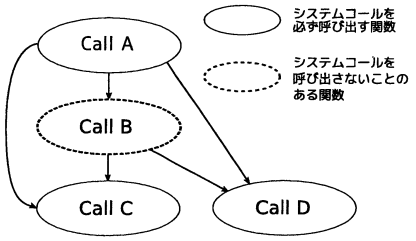

図 7 ライブラリ関数呼び出しが把握できるとは限らないときの動作規則

いる関数の呼び出し順が動作規則にあわなくなるため，検知できる．

攻撃者が関数の戻りアドレスを書き換え，ユーザ関数に制御を移す攻撃について考える．不正に呼び出されたユーザ関数からライブラリ関数が呼び出されるとき，コールスタック履歴にライブラリ関数呼び出し時の記録が残る．そして，コールスタック履歴に残っている関数の呼び出し順が動作規則と一致しないため検知できる．

5.4 提案システムに対する攻撃

攻撃対象が提案システムにより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．

提案システムではスタックに積まれた ebp の値を用いてユーザ関数あるいはライブラリ関数への戻りアドレスを取得し，それを基に動作規則との照合を行いプログラムの動作を判定する．そのため，攻撃者が正常時に取りうるシステムコール，およびスタックに積まれた ebp レジスタの値，戻りアドレスをすべて再現し，さらに，libelem で定義されている関数を動作規則に定義されている順番で呼び出したなら検知することができない．しかしこれらを完璧に再現し，目的のシステムコールを発行することはきわめて困難である．なお，本稿では実装を行っていないが，e-NeXSh のようにメモリレイアウトのランダム化を行うことで攻撃難度をさらに上げることができる．

攻撃者はユーザ空間に存在するコールスタック履歴を書き換えることでスタック偽装の手間を省こうと考えるであろう．しかし，コールスタック履歴を偽装できたとしてもライブラリ関数の実行順まで完全に偽装しつつ，目的の攻撃を行うことは非常に困難である．

攻撃者が環境変数 LD_PRELOAD を書き換えた場合，本手法で作成したライブラリ libelem がリンクされなくなり，ライブラリ関数のフックが行えない．しかし，カーネルではシステムコールをフックしたとき，監視しているプログラムのライブラリ関数呼び出し履歴が取得できなくなるため，動作規則との照合に失敗する．そのため，攻撃者が意図したシステムコールを呼び出させることはできない．

以上のこととから，攻撃対象が提案システムで守られていることを事前知られても，攻撃を成功させるのは困難である．

5.5 提案システムの問題点

本システムでは関数ポインタにより呼び出される関数の特定をしていない．このため，動作規則との照合で関数ポインタを用いた関数呼び出しを確認する場合，すべての関数が呼び出される対象としている．これはプログラムの正常な動作を限定するという本来の目的にそぐわないことであり，検知精度の低下につながる．今後，動作規則作成時に関数ポインタにより呼び出される関数の候補を解析して検知に反映できるようにしていく予定である．なお，監視オーバヘッドの測定に用いたプログラム wc には関数ポインタを用いた呼び出しはなかったが，inetd にはあった．

6. まとめと今後の課題

libelem によりライブラリ関数をフックし，プログラムがライブラリ関数を呼び出したときの状態を把握することで，きめ細かなプログラムの実行を知ることができる．また，ライブラリ関数フック時にフラグを立てることよって，システムコールの発行がライブラリ関数から行われているかを確認できる．branching factor の評価を行うことで，提案システムによってプログラムに許される動作をより限定できることを確認した．

また，libelem で収集した情報は libelem で確保したメモリ領域に記憶する．そしてカーネルがプログラムの動作確認を行うときに読み込むことで，オーバヘッドの削減ができる．今後はこのメモリ領域を攻撃者から守る手段を実装していく予定である．

今回はポインタを用いた関数の呼び出しがあった場合，呼び出し先を特定していない．今後，関数ポインタで呼び出される関数の候補を静的解析により見つけ出し，呼び出し先を限定できるようにする予定である．

本稿での検知精度に関する評価は branching factor のみであったが，今後，実際にマルウェアを用いた評価を行う．

参 考 文 献

1) C.Warrender, S.Forrest, B.Pearlmutter and B.Pearlmutter: Detecting Intrutions Using System Call: Alternative Data Models, *Proc. 2001 IEEE Symposium on Security and Privacy*,

Oakland, pp.156–168 (2001).
2) H.H.Feng, O.M.Kolesnikov, P.Fogla, W.Lee and W.Gong: Anomaly Detectioin Using Call Stack Information, IEEE Symposium on Security and Privacy, Berkeley, CA, pp. 62–77 (2003).
3) Wagner, D. and Dean, D.: Interusion Detection via Static Analysis, *IEEE Symposium on Security and Privacy*, pp.144–155 (2001).
4) 安部 洋, 大山恵弘, 岡 端起, 加藤和彦：静的解析に基づく侵入検知システムの最適化, 情報処理学会論文誌 Vol. 45, No. SIG 3(ACS 5), pp.11-20 (2004.3).
5) S.Kc, G. and D.Keromytis, A.: e-NeXSh: Achieving an Effectively Non-Executable Stack and Heap via System-Call Policing, *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, Tucson, AZ (December 2005).
6) 神山貴幸, 大山恵弘：コールスタック情報を利用したモデル分割に基づく異常検知システム, コンピュータシステム・シンポジウム (ComSys 2008), pp.25-34 (2008.11).