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　38–49 (Mar. 2010)

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

槙本裕司[†1,*1]　齋藤彰一[†1]　古屋雄介[†1]
白井宏憲[†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発行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，正常な状態を偽装した攻撃による耐性が十分でない．本研究では，ライブラリ関数呼び出し時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侵入防止システム Belem を提案する．さらに，Belem を Linux 上に実装して性能評価を行い，検知精度の向上と攻撃検知の有効性を示した．

Belem: Intrusion Prevention System Based on Monitoring Library Function Call

Yuji Makimoto,[†1,*1] Shoichi Saito,[†1]
Yusuke Furuya,[†1] Hironori Shirai,[†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 information and past record of system call. In this paper, we propose a novel intrusion prevention system, named Belem, by monitoring library function call. Belem checks a call stack before a library function executes and completely observes process behavior. We implemented Belem on Linux, and evaluated it.

1. はじめに

セキュリティホールの発見から修正されるまでの無防備な期間をねらう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ある．異常検知方式とは，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，正常な動作規則を確認しながらプログラムを実行することによって，異常な関数呼び出しとシステムコールの発行を検出する方式である．この方式は，未知のセキュリティホールに対しても，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．本稿では，異常検知方式に基づく新しい侵入防止システムの提案と評価について述べる．

多くの攻撃では，バッファオーバフロー攻撃などによってプロセスの制御を奪い，攻撃者が送り込んだコードや libc などのシステムに元からあるコードを利用して攻撃を行う．このため，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呼び出しやシステムコール発行を行う．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．これに対して，多くの異常検知方式では，プログラムが正常に動作していたときの情報や，プログラムのソースコードと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には，プログラムがシステムコールを発行したタイミングに，システムコールフックを用いる手法が多い[1)–3)]．これらの手法は，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しない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，すでに終了して

†1 名古屋工業大学
Nagoya Institute of Technology
†2 京都大学
Kyoto University
*1 現在，三菱電機株式会社
Presently with Mitsubishi Electric Corporation

いる関数は分からないためである．そのため，システムコール発行時だけの監視では，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（Mimicry Attack [1]）を検知できない可能性がある．

この問題を解決するために本稿では，ライブラリ関数呼び出し監視による侵入防止システム **Belem**（Belem is Effective Library Executing Monitor）を提案する．Belem は，ライブラリ関数の実行順に基づく動作規則と，ユーザライブラリ（libelem）とその対応カーネルで構成する．動作規則に，実行ファイルの解析によって得られる正常なライブラリ関数呼び出し順を用いることにより，異常なライブラリ関数呼び出しを検知する．これにより，既存のシステムコール発行順による異常検知システムと比較して検知精度の向上が可能である．なぜなら，システムコールはライブラリ関数を経由して呼び出されており，かつ，システムコールを呼び出さないライブラリ関数もある．このことから，ライブラリ関数呼び出し順は，システムコール発行順を含むより詳細な実行を記録できるためである．そこで Belem は，libelem によっ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を確認し，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の実行を記録する．これにより，ライブラリ関数レベルでの詳細なプロセス監視を実現する．さらに，システムコールが正しい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，システムコールをライブラリ関数を介さずに直接呼び出す攻撃を防止する．

本稿では 2 章でプロセス監視法と関連研究について述べる．3 章で我々の提案システム Belem について述べる．4 章で実装について述べ，5 章で評価を述べる．6 章で Belem への攻撃についての考察を述べる．7 章でまとめと今後の課題を述べる．

2. プロセス監視法と関連研究

既存の侵入検知システムおよび侵入防止システムにおける異常検知手法についてまとめ，これらのプロセス監視法とその問題点について述べる．

2.1 概　　要

侵入検知・防止システムにおける異常検知手法は，正常時の動作を表した動作規則に基づき，対象プロセスの実行を監視する手法である．侵入検知・防止システムは動作規則の作成法とプロセスの監視法により分類することができる．**表 1** に，本章で述べる関連研究の分類を示す．

動作規則の作成法は「学習」と「静的解析」に分類できる．「学習」は監視対象のプログラムを実際に実行させてその動作を解析することで動作規則を作成する．しかし，学習時に現れなかった動作がすべて異常と判断されるために，完全な学習が必要である．しかし，すべての動作を学習させることは困難であり，誤検知（false positive）が発生する可能性がある．一方「静的解析」は，ソースコードあるいは実行ファイルを解析することで，動作規則を作成する．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，プロセス監視時に false positive が発生しないという特徴がある．

表 1　関連研究の分類
Table 1　Classification of related researches.

システム	動作規則作成法	プロセス監視法
Feng ほか[2]	学習	システムコール発行監視
Warrender ほか[4]	学習	システムコール発行監視
Wagner ほか[1]	静的解析	システムコール発行監視
阿部ほか[3]	静的解析	システムコール発行監視
e-NeXSh [5]	学習	ライブラリ関数呼び出し監視

次に，プロセス監視法は「システムコール発行監視法」と「ライブラリ関数呼び出し監視法」に分類できる．「システムコール発行監視法」は，監視対象プロセスが発行したシステムコールに基づいて動作規則の確認を行う．「ライブラリ関数呼び出し監視法」は呼び出したライブラリ関数に基づいて動作規則の確認を行う．以下，関連研究におけるプロセス監視法について，システムコール発行監視法とライブラリ関数呼び出し監視法について述べる．

2.2 システムコール発行監視法

システムコール発行監視法は，発行されたシステムコールに基づいて動作規則と比較する手法である．この手法をとるシステム[1)–3)] は多い．以下，システムコール発行監視法の概要と問題点について述べる．

2.2.1 既存システムの概要

Wagner らの手法[1] では，監視対象プロセスがシステムコールを発行するたびにシステムコールの発行順が動作規則に従っているかを確認する．しかし，システムコールの発行順以外の情報を用いないため，プロセスの実行状態を唯一に特定することが困難になるという問題がある．そのため，考慮すべき状態の数が増加し，状態遷移先の決定のためのオーバヘッドが大きい．また，プロセスの実行状態を特定できないことは，許容するプロセスの状態遷移が増え，正常な実行を偽装した攻撃が容易になる．

一方，Feng らの手法[2] や阿部らの手法[3] では，監視対象プロセスがシステムコールを発行したときのコールスタックを確認することで，プロセスの実行状態をより正確に特定する．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その戻りアドレスが

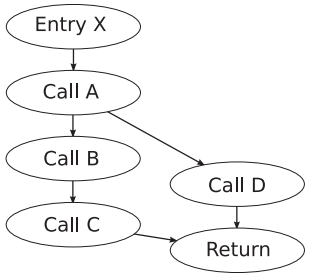


図 1　関連研究の動作規則例
Fig. 1　Example of action rule on related researches.

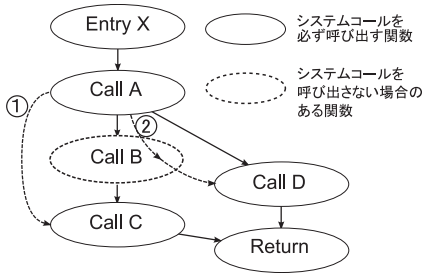


図 2　関連研究の動作規則の問題点
Fig. 2　Problem of action rule on related researches.

コールスタックに積まれる．その戻りアドレスを調べることで，当該システムコールをどの関数が呼び出しているかを確認し，プロセスの実行状態を特定する．さらに，システムコールが前回発行されたとき以降に呼び出されている関数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かを特定する．これにより，システムコールの発行順では特定できない実行状態を把握している．しかし，Feng らと阿部らのどちらの手法においても，システムコール発行時のコールスタックから確認できる関数は，システムコールが発行されたときに実行途中の関数に限られる．つまり，これらの手法では，システムコールを発行することなく終了した関数を確認することはできない．

2.2.2　問　題　点

システムコール発行監視法の問題点について，阿部らの手法を例に述べる．阿部らの手法の正常な動作規則は，関数単位で関数の呼び出し順を定義している．関数 X についての動作規則が**図 1** のように表されていた場合を考える．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．

監視対象プロセスで 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 が終了した後に関数 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．ここで，関数 B がつねにシステムコールを発行する関数でなかった場合，阿部らの手法では次のよう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，プロセス監視時に関数 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，阿部らの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義である．つまり，静的解析で作成された動作規則はプロセスの詳細な状態遷移を表すことができるが，監視しているプロセスの状態の把握をそれと同程度に詳細に行えないため，許容する遷移が増加する．

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 を呼び出す攻撃を考える．このとき関数 B がシステムコールを呼び出していた場合は，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ない場合，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．

2.3　ライブラリ関数呼び出し監視法

ライブラリ関数呼び出し監視法は，呼び出されたライブラリ関数に基づく情報が動作規則と一致しているかを確認する手法である．システムコール発行監視法と比較して，詳細な状態遷移を監視できるため，動作規則を厳密にできる．この結果，図 2 の遷移 ② のような異常遷移を防止することができる．しかし，この監視法を動作規則の厳密化に適用している異常検知システムはない．以下で述べる，e-NeXSh[5] は，ライブラリ関数呼び出し監視をシステムコール呼び出しの正常確認にのみ使用する．以下，e-NeXSh の処理概要と問題点について述べる．

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を監視する．その方法は，まずライブラリ関数をフックし，コールスタックの確認を行う．コールスタック内の戻りアドレスが正当なアドレス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を行う．次に，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専用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，そのうえ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．以上により，攻撃者が，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発行することを防止する．

しかし，e-NeXSh によるライブラリ関数確認処理では，そのつど，専用システムコールを発行するため，確認処理のオーバヘッドは非常に大きい．これに対して，確認処理の対象となるライブラリ関数を攻撃者が利用する確率が高いものに限定し，監視処理のオーバヘッドの増加を抑えている．このために，システムコールを呼び出さないライブラリ関数の監視ができず，プロセス監視精度はシステムコール監視法と同程度である．よって，正常な動作に偽装した攻撃を検知できない可能性がある．

3. 提 案 手 法

新しい侵入防止システム Belem を提案する．Belem は，前章で述べた既存方式の問題点を解決するライブラリ関数呼び出し監視法による侵入防止システムである．システムコール呼び出し順を包含するライブラリ関数呼び出し順と動作規則を用いて，より詳細にプロセスの実行状態を監視する．よって，2.2.2 項で述べた，監視できなかったライブラリ関数に起因する実行状態把握が困難になるという問題は発生しない．

また，Belem は，バッファオーバフロー攻撃などを起点とするコードインジェクションや Return-into-libc，Mimicry Attack による攻撃を対象とする．なお，バッファオーバフロー攻撃そのものは検知の対象としない．これは，バッファオーバフロー攻撃の後の各種活動による不正なライブラリ関数とシステムコールを Belem が検知できるためである．以下，Belem の各特長について述べる．

3.1 監視対象のライブラリ関数

監視対象のライブラリ関数の数は，多ければ多いほど実行状態を正確に特定できる．しかし，監視対象ライブラリ関数の増加は監視オーバヘッドを増加させるため，適切に設定する必要がある．既存システムの e-NeXsh では，攻撃者が使用する確率の高いライブラリ関数を監視対象としている．しかし，どのようなプログラムにおいても，そのようなライブラリ関数が高頻度で利用されるわけではないため，2.3 節で述べたように正確な実行監視には不十分である．そこで，Belem では，システムコール発行監視法で監視できるシステムコールをすべて網羅するために，システムコールを発行する可能性のあるすべてのライブラリ関数を監視対象とする．これにより，すべてのシステムコール発行の元となるライブラリ関数を監視でき，かつシステムコールを発行しなかった場合についても監視できる．以上から，システムコール監視法以上の実行状態把握を実現する．

3.2 ユーザ空間への履歴保存

監視対象ライブラリ関数の増加は，監視オーバヘッドを増加させる．e-NeXSh では，専用システムコールによるライブラリ呼び出し記録を行う．このため，ライブラリ関数呼び出しごとにシステムコールが発行されることになり，そのオーバヘッドはライブラリ関数呼び出しだけの場合と比較して大きい．Belem では，このオーバヘッド増加を抑えるため，コールスタック履歴をユーザメモリ空間内に保存する．この保存のためのオーバヘッドは，システムコール発行と比較して十分に小さいことは明らかである．これにより，すべてのシステムコール発行元となるライブラリ関数監視を低負荷で実現する．

3.3 利用可能システムコールの制限

Belem におけるシステムコールの制御について述べる．Belem での実行監視はライブラリ関数単位であるため，システムコール単位での動作規則はない．このため，ライブラリレベルの状態遷移が正常な場合においても，ライブラリ内部の脆弱性に起因する攻撃が発生する可能性がある．Belem では，この問題に対応するために，次の 2 点によりシステムコール発行を監視する．

(1)　各ライブラリ関数の利用するシステムコールを，あらかじめ関数単位で規則化する．システムコール発行時に，当該システムコールの利用が呼び出し元のライブラリ関数の規則に含まれるか否かを確認する．

(2)　システムコール実行を，Belem を介したライブラリ関数実行中に限定する．

これらの確認の前提条件として，システムコール発行には，コールスタック履歴に必要なライブラリ呼び出しが記録されており，Belem カーネルにより正常であると確認されることがある．

前提条件により，ライブラリ関数レベルでの実行状態が正常であることを確認できる．(1) により，当該ライブラリ関数が使用しないシステムコールを実行することができない．(2) により，攻撃者がスタックを正常であるかのように偽装したうえで Belem を介さずに直接 libc の関数を呼び出したとしても，システムコールを実行することができない．これらか

ら，発行されたシステムコールは，ライブラリ関数レベルでの実行状態が正常であり，かつ，当該ライブラリが使用するものであることが確認できる．

4. 実　　装

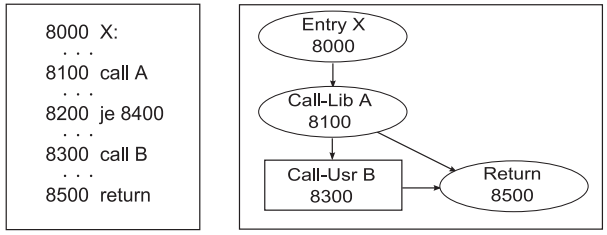
Belem の Linux への実装の詳細について述べる．Belem は，Linux カーネルを改造した**Belem カーネル**と，ライブラリ関数をフックするためのライブラリ **libelem** の 2 つで構成される．Belem カーネルは，Linux カーネルのシステムコールエントリ処理部（entry.S）をフックすることで実現されており，システムコールと動作規則，コールスタック履歴の確認を行う．また，libelem は，ライブラリ関数をフックしコールスタック履歴を作成する．

本章では，まず，プロセスの正常実行を規定する動作規則の概要について述べる．次に，Belem の全体の処理の流れを示した後で，Belem におけるライブラリ関数呼び出し監視法について，コールスタック履歴の作成と動作履歴の確認を用いて詳細に述べる．

4.1　動作規則の概要と作成手法

動作規則の内容と作成手法について述べる．動作規則は，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記録したものである．関数 1 つの動作規則を図を用いて表した例を**図 3** に示す．図 3 (a) は，関数 X のアセンブリコードの例である．これを基に作成した関数 1 つの動作規則が図 3 (b) である．各関数の動作規則は，始点である Entry ノード，終点である Return ノードと，ユーザ関数呼び出しとライブラリ関数呼び出しを表すノードで構成される．

動作規則の作成は，監視対象プログラムの実行ファイルを逆アセンブルし，その結果を解析することで行う．逆アセンブル結果を各関数単位に分割して関数単位とする．関数単位の逆アセンブル結果について，関数呼び出しを抽出してノードを決定する．続いて，ジャンプ命令と分岐命令を抽出し，各ノード間の状態遷移を決定する．この際，各ノードには呼び出し元アドレスを併せて記録する．呼び出し元アドレスによって，単一の関数からの複数の同一関数呼び出しの区別が可能となり，遷移先ノードをただ 1 つに特定する．以上の手順で関数単位の動作規則を作成する．これをすべての関数に対して行うことで，プログラムの動作規則を作成する．作成された動作規則は，実行ファイルとマージされて 1 ファイルとし，電子署名を施す．これにより，プログラムロード時に対象プロラグムと動作規則を明確に関連付けることができ，さらに改竄されていないことを確認できる．



(a) アセンブリコード　(b) 動作規則の概念図
(a) Assembly code.　(b) Schematic action.

図 3　動作規則の作成
Fig. 3　Generation of action rule.

動作規則のサイズを**表 2** に示す．対象アプリケーションは 5 章で評価に使用する wc と inetd である．表 2 の実行ファイルとは動作規則の作成元のファイル全体を示す．text セクションは実行ファイルの中に含まれるセクションで，ユーザが作成した実行コードを含むセクションである．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．今後，動作規則のサイズを小さくするように，フォーマットの改良を行う予定である．

4.2　全体の処理の流れ

Belem の全体構成を**図 4** に示す．図 4 は，ユーザプロセスが read 関数を呼び出したときの処理の流れを示している．ユーザプロセスが read 関数を呼び出すと，環境変数 LD_PRELOAD により 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，dlsym 関数を用いて 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），Belem カーネルがコールスタック履歴を参照し，libelem によりライブラリ関数がフックされているかの確認と，動作規則の確認を行う（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，ユーザプログラムに戻る（⑩）．

表 2　動作規則のサイズ（KB）
Table 2　Size of action role (KB).

アプリケーション	実行ファイル	text セクション	動作規則
wc	85	9	13
inetd	30	11	18

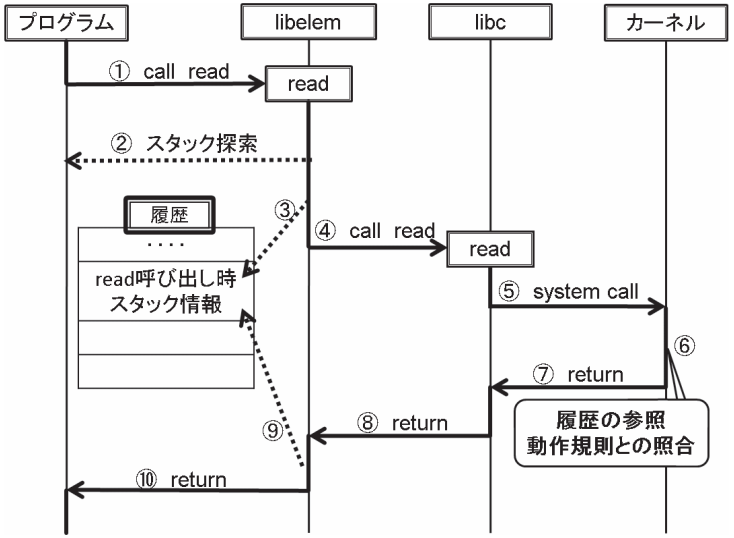
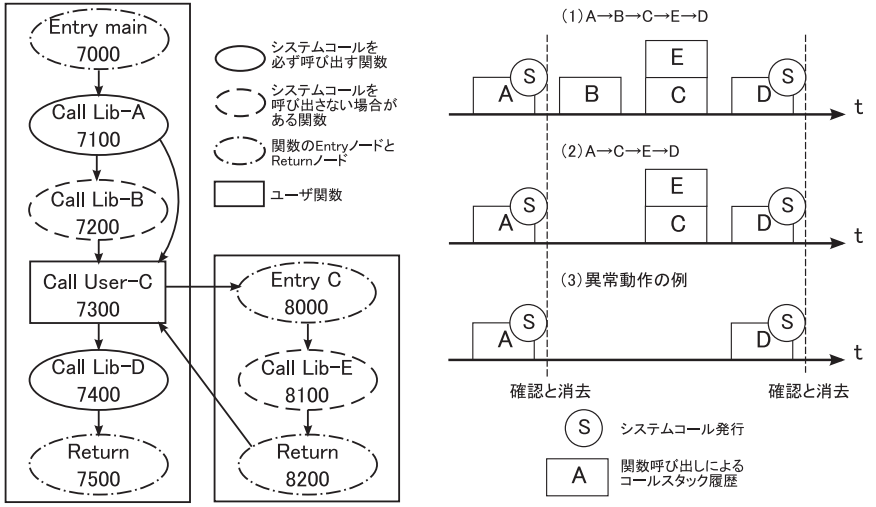


図 4　Belem の処理の流れ
Fig. 4　Process flow of Belem.



(a) 動作規則の例
(a) Example of action rule.

(b) コールスタック履歴の例
(b) Examples of call stack history.

図 5　コールスタック履歴の変化と動作規則の確認
Fig. 5　Change of call stack history and confirmation of action rule.

4.3　コールスタック履歴と動作の確認

プロセスの動作確認のために，Belem ではライブラリ関数呼び出し時にコールスタック履歴の作成と，システムコール発行時に動作規則とコールスタック履歴の確認とシステムコールの種類の確認を行う．以下，これらの処理内容について述べる．また，シグナル処理と fork/exec 処理についても述べる．

4.3.1　コールスタック履歴の作成

libelem は，ライブラリ関数呼び出しをフックした後，コールスタック履歴を作成する（図 4 の ③）．**図 5** にコールスタック履歴の作成例を示す．図 5 (a) に示す動作規則例に基づくコールスタック履歴が (b) である．(a) の動作規則例は，main 関数とライブラリ関数 A と B と D，ユーザ関数 C とそこから呼ばれるライブラリ関数 E によって構成される．ライブラリ関数 A と D は必ずシステムコールを発行するが，その他の各関数はシステムコールを発行するか否かは決まっていない．(b) のコールスタック履歴の例には，図 5 (a) の動作規則例から考えられる動作パターンは (1) A → B → C → E → D と (2) A → C → E → D の 2 種類を示す．なお，(3) は異常動作の例であり後述する．

コールスタック履歴は，ライブラリ関数呼び出しごとに追記し，次のシステムコール発行時に Belem カーネルが動作規則と一致確認し，その後に消去する．このため，図 5 (b) に点線で示した「確認と消去」の間が，1 回の動作規則確認対象区間である．各ライブラリ関数呼び出しにおけるコールスタック履歴の内容は，1) main 関数から現在に至るまでの関数呼び出し元アドレス，2) ライブラリ関数を実行中を示すフラグの 2 要素で構成される．第 1 要素は，呼び出し時のライブラリ関数のコールスタックから戻りアドレス群を抽出したものである（なお，図 5 (b) では簡潔に関数名で記した）．たとえば，関数 A は，main 関数から直接呼び出されるため，コールスタック履歴はただ 1 つ関数 A の戻りアドレスである．また，関数 E は関数 C から呼び出されるため，コールスタック履歴に関数 C と E の戻りアドレスを記録する．このようにして，すべてのライブラリ関数の呼び出し時にコールスタック履歴を作成する．なお，同じライブラリ関数が同じアドレスから連続して呼び出された場合には，履歴の作成を省略する．これは，ループなどによるライブラリ関数の繰返し呼び出しが動作規則で成立する場合には，コールスタック履歴数の増加を防止するために，

作成時点で動作規則の判断をしている[⋆1]．次に，第 2 要素の実行中フラグは，3.3 節の監視 (2) のために使用する．実行終了の記録には，LD_PRELOAD によるライブラリ関数フックがライブラリ関数終了後にも行われることを利用して記録する（図 4 の ⑨）．

4.3.2　動作規則の確認

Belem カーネルは，システムコールをフックした後，動作規則の確認処理を行う（図 4 の ⑥）．以下，動作規則の確認処理の流れについて述べる．

まず，Belem カーネルは監視対象プロセスごとに前回までの動作確認完了ノードを記憶する．確認処理は，この前回完了ノードを起点に行う．以下の説明では，図 5 の Lib-A が動作確認完了ノードとし，コールスタック履歴は (1) の場合とする．動作確認は，まず，前回確認完了ノードである Lib-A を確認ノードとし，ここからの遷移先 Lib-B と User-C を抽出し，コールスタック履歴中の 1 つ目の履歴と比較する．この場合は，Lib-B が一致することが判明し，Lib-B を次の確認ノードとする．もし，一致しない場合は，次の遷移先（図 5 の例では User-C）を確認する（コールスタック履歴 (2) の場合に該当する）．続いて，Lib-B からの遷移先である User-C を抽出する．ユーザ関数であるため，その戻りアドレス（7300）を記録し，User-C の Entry ノードを確認ノードとする．以下，同様に繰り返すことで Lib-E に到達する．ここで，記憶していた戻りアドレス（7300）と Lib-E を，コールスタック履歴 User-C → Lib-E と比較し一致を確認する．Return ノードの場合は，記憶していた戻りアドレスノードへ確認ノードを移す．以降，これを現時点のノードに一致するまで繰り返し実施する．

また，抽出したノードがライブラリ関数であるがコールスタック履歴に一致しない場合は，当該確認処理は失敗となる．すべての確認処理が失敗となった場合には，異常と判断する．たとえば，図 5 (b) の (3) の異常動作の例では，Lib-A の次に Lib-D が現れている．しかし，動作規則の Lib-A からの探索は，Lib-B と User-C → Lib-E というライブラリ関数までで完了し，Lib-D は含まれない．よって，この例は異常と判断する．この動作を異常と判断できる点が，Belem におけるライブラリ関数呼び出し監視法の特長である．

このようにして，コールスタック履歴に一致する動作規則を探索することで，プロセスが正常に動作していることを確認する．もし異常と判断された場合には，発行されたシステムコールに代わって kill システムコールを監視対象プロセスに対して実行し，プロセスを終了させる．

⋆1 ループの繰返し回数などの動的に決定される情報は，動作規則に含まれない．

4.3.3　ライブラリ関数から発行されたシステムコールの確認

3.3 節で述べたように，Belem では，ライブラリ関数ごとに発行できるシステムコールを制限する．利用できるシステムコールは，動作規則作成時に静的に解析し，各ライブラリが使用するすべてのシステムコールと発行元アドレスを登録する．これは，無関係なシステムコールと無効な発行元アドレスからのシステムコールを検出するためである．

プロセス実行中にシステムコールが発行されたとき，システムコール発行元のライブラリ関数とシステムコールの対応が正しいか否かを確認する．この対応が正しくない場合は異常な動作であると判断する．

4.3.4　シグナル対応

シグナルハンドラについても他のユーザ関数と同様に，事前に動作規則を作成する．監視しているプロセスに対してシグナルが送られた場合，シグナルハンドラを検索し当該シグナルハンドラのアドレスを取得する．続いて，シグナルハンドラに対応した動作規則の Entry ノードから確認を行う．

4.3.5　fork/exec 対応

監視対象プロセスが fork を実行した場合は，子プロセスも監視対象プロセスとする．この場合，動作規則とコールスタック履歴は，各プロセスで個別に管理する．また，execve システムコールを呼び出した場合にはいったん監視を打ち切る．新たに実行するプログラムが動作規則を有していれば，監視対象とする．この場合も，動作規則とコールスタック履歴は，各プロセス独自となる．なお，親プロセスが非監視対象プロセスであっても，子プロセスが動作規則を有していれば子プロセスだけ監視対象プロセスとなる．

5. 評　　価

Belem の評価について述べる．評価は，動作規則におけるライブラリ関数呼び出しノード間の遷移数の平均（以下，平均遷移数という）を用いた検知精度の評価，監視オーバヘッドの測定および単純な攻撃コードによる検証である．評価環境は Pentium 4（3.2GHz）（HT 無効）上で動作する Fedora Core 5 である．また，Belem は Linux カーネル 2.6.17.8 を変更して実装した．評価アプリケーションとして，ファイルユーティリティの wc と，ネットワークサーバの inetd を用いる．

5.1　平均遷移数による検知精度の評価

動作規則の検知精度の評価には平均遷移数を用いる．平均遷移数は動作規則に含まれる遷移の総数をノードの総数で割った値である．平均遷移数が小さいほど次に遷移可能なノード

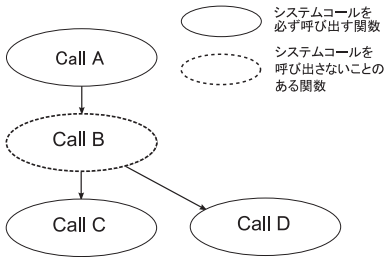


図 6　完全把握方式（Belem）の動作規則
Fig. 6　Action rule of completely grasp method (Belem).

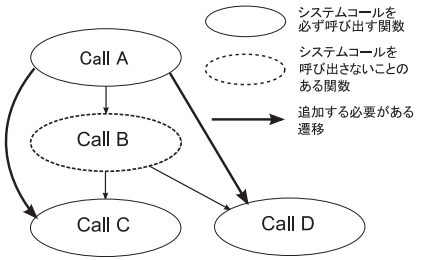


図 7　部分把握方式（e-NeXSh など）の動作規則
Fig. 7　Action rule of partially grasp method (e-NeXSh).

が限定されるため，プロセスの実行を厳しく制限することができ，検知を回避する攻撃が困難になる．Belem により，ライブラリ関数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評価する．比較対象としてライブラリ関数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平均遷移数を用いる．以下，ライブラリ関数呼び出しが完全に把握できる方式を**完全把握方式** といい，すべてのライブラリ関数呼び出しを把握できない方式を**部分把握方式**という．完全把握方式は，Belem が該当する．部分把握方式は，システムコール発行監視法による異常検知システムと，2.3 節で述べたようにシステムコール発行監視法と同程度のプロセス監視精度である e-NeXSh が該当する．

部分把握方式の動作規則の作成法について述べる．まず，ライブラリ関数を，システムコールを必ず呼び出すものと，呼び出さない場合のあるものに分類する．システムコールを呼び出さない場合のあるライブラリ関数は，呼び出しが把握できない場合の遷移を動作規則に追加する．ここで，完全把握方式である Belem の動作規則が **図 6** のように表されており，関数 B がシステムコールを呼び出さない可能性があるライブラリ関数である場合について考える．部分把握方式では，関数 B がシステムコールを発行せずに終了すると，関数 B が呼び出されたか否かを確認できない．したがって，部分把握方式における遷移は，関数 B の直前のノードから関数 B の直後のノードに直接遷移する場合と同様となる．この場合の動作規則を**図 7** に示す．関数 A から関数 C への遷移と関数 D への遷移が増えていることが分かる．

部分把握方式の平均遷移数を**表 3** に，完全把握方式の平均遷移数を**表 4** に示す．表 3 と表 4 から，完全把握方式の平均遷移数と遷移の総数が，部分把握方式の値よりも小さく，動作規則の遷移がより限定されていることが分かる．以上から，Belem のような完全把握方式は，部分把握方式よりも攻撃の自由度が低く，安全であるといえる．

表 3　部分把握方式（e-NeXSh など）の平均遷移数
Table 3　Average number of state transitions of partially grasp method (e-NeXSh).

アプリケーション	平均遷移数	遷移の総数	ノードの総数
wc	1.57	367	234
inetd	2.01	783	389

表 4　完全把握方式（Belem）の平均遷移数
Table 4　Average number of state transitions of completely grasp method (Belem).

アプリケーション	平均遷移数	遷移の総数	ノードの総数
wc	1.33	312	234
inetd	1.48	574	389

5.2　監視オーバヘッドの測定

ファイルユーティリティの wc は，ファイルを走査するたびにライブラリ関数を呼び出すが，システムコール発行は少ないという特徴がある．ライブラリ関数呼び出し回数が約 1,500 万回（15 MB のファイルの場合）に対して，システムコールは約 1,000 回である．一方，ネットワークサーバの inetd は，1 接続あたりのライブラリ関数呼び出しとシステムコール発行が約 1,000 回ずつとほぼ同じである．これらアプリケーションの結果により，ライブラリ関数呼び出し監視法によるライブラリ関数フックの影響を評価する．

wc および inetd を監視したときのオーバヘッドを**表 5** に示す．wc は 15 MB のファイルを対象とした場合の測定結果で，inetd は別ホストから 1,000 回接続を行った場合の測定結果である．なお，監視対象としたのは inetd のみであり，inetd から起動されるプロセスは

表 5　監視オーバヘッド
Table 5　Observation overhead.

アプリケーション	通常実行	ライブラリ関数フックのみ	Belem 全処理
wc	1.69 (1.00)	2.26 (1.34)	3.33 (1.97)
inetd	2.53 (1.00)	2.60 (1.03)	3.39 (1.34)

単位は秒，括弧内は通常実行に対する増加割合

監視対象ではない．各アプリケーションにおいて，測定を行ったのは，1) 通常環境での実行の場合，2) libelem を用いてライブラリ関数のフックのみを行った場合，3) Belem による全処理（ライブラリ関数フックと動作規則と確認実施）の 3 種類である．

ライブラリ関数のみフックの結果から，wc の増加割合が inetd と比べて大きいことが分かる．これは，先に述べたように wc のシステムコール発行数に対するライブラリ関数呼び出し数が大きいため，ライブラリ関数呼び出し監視による負荷によるものである．

以上から，wc のように，システムコール発行数に対するライブラリ関数呼び出し数が非常に多い場合は，監視オーバヘッドが大きくなる．しかし，inetd のようにシステムコール発行数とライブラリ関数呼び出し数が同程度であれば，監視オーバヘッドは約 30%であり，昨今の高速な計算機であれば許容範囲内であると考える．

5.3　攻撃コードによる検証

文献 5) を参考に単純な攻撃コードを作成し，以下の 2 つの場合において Belem の攻撃防止性能を検証した．なお，評価で用いた攻撃コードは次の 2 種類であり，これらのプログラムを**図 8** と**図 9** に示す．

- 戻りアドレスを偽装しない攻撃
- 戻りアドレスを偽装する攻撃

これらの評価プログラムは関数 func の戻りアドレスをライブラリ関数 execve のアドレスに書き換えてシェルを起動させるものである．ライブラリ関数 execve の呼び出しおよびそこからの execve システムコールの発行は異常な動作によるものであり，動作規則にはこの呼び出しが定義されていないものとする．以下，戻りアドレスの偽装の有無による侵入防止手順について述べる．

5.3.1　戻りアドレスを偽装しない攻撃

バッファオーバフローにより関数の戻りアドレスを書き換え，任意のライブラリ関数を呼び出す攻撃について考える．評価プログラム 1（図 8）を実行すると，関数 func 後に実行されるライブラリ関数 execve から，execve システムコールが発行される．Belem カーネルは，このシステムコールをフックし，コールスタック履歴にライブラリ関数 execve が libelem にフックされて実行中であるという記録が書かれていることを確認する．確認後に Belem カーネルはコールスタック履歴の確認を行う．しかし，コールスタック履歴内のライブラリ関数 execve の戻りアドレスがユーザ関数内のアドレスではないことが分かる．このことから，ライブラリ関数 execve が不正な手段で呼び出されていると判断し，execve システムコールを処理せず，プロセスの実行を強制的に終了させ，不正実行を防止する．

```
#include <unistd.h>
char *argv[] = { "/bin/sh" };

int func(void){
        void **fp = (void **) &fp;
        fp[2] = execve;
        fp[4] = argv[0];
        fp[5] = argv;
        fp[6] = 0;
        return 0;
}


int main(void){
        return (func());
}
```

図 8　評価プログラム 1
Fig. 8　Program 1 for evaluation without return address camouflage.

5.3.2　戻りアドレスを偽装する攻撃

攻撃用ライブラリ関数 execve の戻りアドレスをプログラム中の関数のアドレスに設定した評価プログラム 2（図 9）について考える．評価プログラム 2 は，評価プログラム 1 に戻りアドレスの改竄部分を追加して作成した．

評価プログラム 2 を実行すると，関数 func 後に実行されるライブラリ関数 execve から，execve システムコールが発行される．コールスタック履歴を確認するとライブラリ関数 execve が libelem によりフックされて実行中であるという記録が書かれている．また，コールスタック履歴のライブラリ関数 execve の戻りアドレスはユーザ関数内のアドレスで

```
#include <unistd.h>
char *argv[] = { "/bin/sh" };

int func(void){
        void **fp = (void **) &fp;
        fp[2] = execve;
        fp[3] = func;
        fp[4] = argv[0];
        fp[5] = argv;
        fp[6] = 0;
        return 0;
}

int main(void){
        return (func());
}
```

図 9　評価プログラム 2
Fig. 9　Program 2 for evaluation with return address camouflage.

あるため，この時点では問題ない．次に，ライブラリ関数 execve の呼び出しを動作規則で確認する．しかし，動作規則にはライブラリ関数 execve を呼び出す定義が存在しないため，execve システムコールを処理せず，プロセスの実行を強制的に終了させ，不正実行を防止する．

以上の検証により，Belem で実際に攻撃を検知できることを確認した．また，攻撃者が攻撃コードから直接システムコールを呼び出した場合でも，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らないため検知可能である．

6. 考　　察

本章では，Belem に対する攻撃への対応についての考察を述べる．攻撃者が，攻撃対象プロセスが Belem により守られていると知ったうえで，攻撃する場合の対応について述べる．この場合の攻撃手法は，ライブラリ実行順が変化しない攻撃手法と，Belem を無効化する手法がある．

6.1　実行順が変化しない攻撃

実行順が変化しない攻撃は，次の 2 種に大別できる．攻撃 A）システムコールとライブラリ関数を呼び出さない攻撃，攻撃 B）正常な動作を偽装した攻撃である．以下，これらの攻撃について述る．

6.1.1　システムコールとライブラリ関数を呼び出さない攻撃

システムコール呼び出しとライブラリ関数呼び出しを行わない攻撃は，見かけの実行順は変化せず，Belem が異常な実行を検知できない．しかし，この攻撃は，入出力やネットワーク利用が不可能であるため有効な攻撃は困難である．この攻撃で可能な動作として，高負荷計算や無限ループによるプロセス停止攻撃が考えられる．しかし，プロセスが長時間にわたって高負荷状態を続けた場合，Miracle Linux における MAZE 監視[6] のようにカーネルによって検出することができる．

6.1.2　正常な動作を偽装した攻撃

正常な動作を偽装した攻撃とは，コールスタックやコールスタック履歴を改竄することで，あたかも正常な状態遷移を経ていると Belem に判断させて攻撃プログラムを実行させることである．以下，偽装攻撃に対する Belem の耐性について述べる．

まず，コールスタックを正常状態に偽装した攻撃方法は存在する[7]．このような方式を，コールスタック履歴偽装へと拡張することで，Belem に対応した正常状態を偽装しての攻撃は可能と考える．しかし，そのためには，ライブラリ関数呼び出し順，関数の戻りアドレスと関数呼び出し元アドレスを動作規則と一致させなければならない．そのうえで，動作規則に一致したシステムコールしか呼び出せない．したがって，攻撃に使用できる動作は限られる．

Belem は，ライブラリ呼び出しごとにコールスタック履歴を作成する．コールスタックが前回確認状態から遷移できない状態に偽装された場合，コールスタック履歴が動作規則と一致しないことから異常検出可能である．このため，コールスタック偽装では，直前のライブラリ関数呼び出し以前への遷移を偽装することはできない．次に，Belem ではシステムコール発行ごとにコールスタック履歴の確認と削除を行う．この削除により，コールスタック履歴偽装の対象は，直前のシステムコール発行以後のコールスタック履歴情報に限られる．これ以外の履歴を偽装した場合，動作規則と一致せずに検出可能である．したがって，いかなる場合においても直前のシステムコール以前の状態に戻ることはできない．このため，偽装攻撃における状態遷移がシステムコール間内に限られる．

コールスタック履歴偽装については，コールスタック履歴のアドレスをランダム化する

ことで攻撃者から隠蔽することにより，偽装される危険性を減らすことができる．しかし，コールスタック履歴は，libelem 内部から更新するため，libelem 内部にはコールスタック履歴のアドレス情報が必要である．このアドレス情報を隠蔽するために，Belem では，libelem 自体の配置アドレスのランダム化と GOT 参照[8] を再利用しない手法[9]，さらに，proc ファイルシステムを経由したメモリレイアウト参照の禁止による他プロセスを介した配置アドレス流出を防止する．これらにより，コールスタック履歴を保護する．

以上より，偽装による攻撃は可能であるが攻撃に使用できる動作は限られており，さらに Belem では偽装可能な遷移をシステムコール間やライブラリ間に限定する．これにより，攻撃の難易度は格段に上がる．さらに，メモリレイアウトのランダム化により，戻りアドレスの決定を困難にすることで攻撃難度をさらに上げ，結果として攻撃を防止すると考える．

6.2　保護機構の無効化攻撃

Belem が侵入防止を行うためには，libelem が正しくリンクされ，動作規則が正常であることが必要である．これらを無効化することで，Belem の保護機構を無効にする攻撃が考えられる．以下，これらに対する対応について述べる[9]．

6.2.1　libelem の無効化

攻撃者が環境変数 LD_PRELOAD を書き換えた場合，Belem 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，Belem カーネルにおいてシステムコールをフック後，監視対象プロセスのコールスタック履歴を取得できないため，動作規則の確認に失敗する．このため，攻撃者が意図したシステムコールを呼び出すことはできず，攻撃は失敗する．

6.2.2　動作規則の改竄

動作規則を書き換えることで任意の攻撃を Belem 上で実行可能にする攻撃が考えられる．動作規則を書き換えて，攻撃コードから必要なライブラリ関数を発行できるようにする攻撃である．この攻撃を防止するために，Belem では，動作規則と実行バイナリを連結して 1 ファイルとし，さらに電子署名を施す．電子署名は，実行バイナリのコンパイル実行者もしくは実行バイナリの素性を確認できる者（ディストリビューションの作成者など）が行う．署名確認に必要な公開鍵は，Web サイトによる公開や，コンパイル実行者から直接入手する．これらにより，配布中における動作規則の保護を行う．

また，実行中の改竄を防止するためには，Belem カーネルがプロセスイメージをメモリ空間に展開するときに，動作規則のアドレス空間を書き込み禁止にする．さらに，mprotect() によるアクセス権変更に対して，動作規則のアドレス空間のアクセス権変更を禁止する．これらにより，実行中における動作規則の保護を行う．

7. ま と め

新しいライブラリ関数呼び出し監視法による侵入防止システム Belem について述べた．Belem は，ライブラリ関数をフックしその履歴をユーザメモリ空間に保存することで，低負荷でプロセスの実行状態を詳細に知ることができる．これにより，システムコール発行監視法や既存のライブラリ関数呼び出し監視法と比較して，ライブラリ関数の呼び出し順を詳細に把握できるため，正常な動作を偽装した攻撃をより困難にする．また，システムコールの発行が libelem を介して行われているかを確認し，さらに，ライブラリ関数ごとに実行できるシステムコールを制限する．これらにより，ゼロデイ攻撃や未知のセキュリティホールに対する攻撃から計算機を保護する．

評価においては，通常実行時間の約 30%増加となり，十分に許容できる遅延時間であることを示した．また，平均遷移数の評価により，Belem は部分把握方式と比較して，厳密にプログラムの動作を限定できることを示した．さらに，単純な攻撃例によって，正しく異常を検出できることを示した．

今後の課題として，実際の exploit を使用した実験，関数ポインタと longjump による実行状態の変化への対応がある．今後，動作規則作成時に関数ポインタと longjump 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する．

参 考 文 献

1) Wagner, D. and Dean, D.: Intrusion Detection via Static Analysis, *IEEE Symposium on Security and Privacy*, pp.144–155 (2001).
2) Feng, H.H., Kolesnikov, O.M., Fogla, P., Lee, W. and Gong, W.: Anomaly Detection Using Call Stack Information, *IEEE Symposium on Security and Privacy*, pp.62–77 (2003).
3) 阿部洋丈，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：コンピューティングシステム，Vol.45, No.SIG 3 (ACS 5), pp.11–20 (2004).
4) Warrender, C., Forrest, S., Pearlmutter, B. and Pearlmutter, B.: Detecting Intrutions Using System Call: Alternative Data Models, *Proc. 2001 IEEE Symposium on Security and Privacy*, pp.156–168 (2001).
5) Gavrav, S.Kc. and Angelos, D.K.: e-NeXSh: Achieving an Effectively Non-Executable Stack and Heap via System-Call Policing, *Proc. 21st Annual Computer*

Security Applications Conference (*ACSAC*), pp.288–302 (2005).
6) Miracle Linux Corporation: MIRACLE TECHNOLOGY DAY 2008 夏 (2008). http://www.miraclelinux.com/corp/event_seminar/techday2008_pdf/TechDay08-NGN-v3.pdf
7) Kruegel, C., Kirda, E., Mutz, D., Robertson, W. and Vigna, G.: Automating Mimicry Attacks Using Static Binary Analysis, *14th USENIX Security Symposium*, pp.161–176 (2005).
8) Drepper, U.: How to Write Shared Libraries (2006). http://people.redhat.com/drepper/dsohowto.pdf
9) 古屋雄介，齋藤彰一，松尾啓志：侵入防止システムにおける動作規則保護機構の開発，情報処理学会研究報告システムソフトウェアとオペレーティングシステム，Vol.2009-OS-112, No.7 (2009).

(平成 21 年 7 月 24 日受付)
(平成 21 年 11 月 29 日採録)

槙本　裕司

2007 年名古屋工業大学工学部電気情報工学科卒業．2009 年同大学大学院情報工学専攻博士前期課程修了．同年三菱電機株式会社入社，現在に至る．修士（工学）．

齋藤　彰一（正会員）

1993 年立命館大学理工学部情報工学科卒業．1995 年同大学大学院博士前期課程修了．1998 年同大学大学院博士後期課程単位習得退学．同年和歌山大学システム工学部情報通信システム学科助手．2003 年同講師，2005 年同助教授．2006 年名古屋工業大学大学院助教授，2007 年同准教授，現在に至る．オペレーティングシステム，インターネット，情報セキュリティ等の研究に従事．博士（工学）．ACM，IEEE-CS 各会員．

古屋　雄介

2009 年名古屋工業大学工学部情報工学科卒業．同年同大学大学院情報工学専攻博士前期課程入学，現在に至る．

白井　宏憲

2009 年名古屋工業大学工学部情報工学科卒業．同年同大学大学院情報工学専攻博士前期課程入学，現在に至る．

上原哲太郎（正会員）

1990 年京都大学工学部情報工学科卒業．1995 年同大学大学院博士課程研究指導認定退学．同年同大学院工学研究科助手，1996 年和歌山大学システム工学部講師，2003 年京都大学大学院工学研究科附属情報センター助教授，2006 年京都大学学術情報メディアセンター助教授，2007 年同准教授．システムソフトウェア，システム管理，情報セキュリティ関係の研究に従事．京都大学博士（工学）．IEEE，電気学会，電子情報通信学会，日本ソフトウェア科学会，システム制御情報学会，情報ネットワーク法学会，CIEC 各会員．

松尾　啓志（正会員）

1983 年名古屋工業大学工学部情報工学科卒業．1985 年同大学大学院修士課程修了．1989 年同大学院博士課程修了．同年名古屋工業大学電気情報工学科助手．講師，助教授を経て，2003 年同大学大学院教授．2006 年同大学情報基盤センターセンター長（併任），現在に至る．分散システム，分散協調処理に関する研究に従事．工学博士．電子情報処理学会，人工知能学会，IEEE 各会員．