平成20年度　　　修士論文

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

指導教官
松尾　啓志　教 授
齋藤　彰一　准 教 授

名古屋工業大学　情報工学専攻
平成19年度入学　19417609番

槙本　裕司

目 次

第1章　はじめに **1**

第2章　関連研究 **3**

2.1 正常な動作の定義作成法 3
2.2 プログラムの実行監視法 4
2.2.1 システムコール発行監視手法 4
2.2.2 ライブラリ関数実行監視手法 7

第3章　提案手法 **9**

3.1 ライブラリ関数を呼び出したときに情報を取得 9
3.2 ライブラリ関数の呼び出し確認 10
3.3 フック時の処理の高速化 10
3.4 動作規則の構成 10
3.4.1 動作規則の作成単位 11
3.4.2 動作規則に定義する関数 11
3.4.3 動作規則の作成方法 11

第4章　実装 **12**

4.1 全体構成 12
4.2 動作規則 13
4.2.1 動作規則の作成例 13
4.2.2 動作規則のフォーマット 15
4.2.3 動作規則のサイズ 17

4.2.4　動作規則の管理 18
4.3　ライブラリ関数の呼び出し確認 18
4.4　コールスタック履歴の作成 19
4.5　動作規則との照合 20

第5章　評価 **22**
5.1　監視オーバヘッドの測定 22
5.2　branching factor による評価 24
5.3　攻撃コードによる検証 26
5.3.1　サンプルプログラムの説明 27
5.3.2　フックしないライブラリ関数を利用した攻撃 28
5.3.3　フックするライブラリ関数を利用した攻撃 29
5.3.4　フックするライブラリ関数を利用し，スタックも偽装した攻撃 . 30

第6章　考察 **32**
6.1　検知性能の考察 32
6.1.1　バッファオーバフロー攻撃を行う3つの要素 32
6.1.2　20種のバッファオーバフロー攻撃パターン 35
6.1.3　提案システム上で攻撃パターンを再現したと仮定したときの評価 36
6.1.4　既存のシステムとの比較 36
6.2　検知可能な攻撃 40
6.3　提案システムに対する攻撃 40
6.4　提案システムの問題点 42

第7章　まとめと今後の課題 **44**

謝辞 **45**

第1章

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．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 [4][7][11]．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い．

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) を提案する．本ライブラリをプログラムにリンクすることで，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い．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 章で評価を述べる．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．

第2章

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 [3][4] がある．これら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 [7][11] がある．これらの手法では，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．また，動作規則は実行ファイルあるいはソースコードから自動生成するため，作成に手間がかからない．しかし，動作規則作成時に演算結果や引数などを解析することが困難である．そのため，条件分岐や関数引数が実行ファイルどおりであるかを検証することができず，通常では起こり得ない実行を許可してしまう false negative が起きる可能性がある．

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 [4][7][11] は多い．Wagner らの手法 [7] 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．彼らの手法では，監視対象のプログラム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ていないため，プログラムの実行状態を唯一に特定することが困難である．そのため，考慮すべき状態の数が増加してしまい，オーバヘッドが非常に大きくなる．また，プログラムの実行状態を特定できないことは，許容するプログラムの実行状態を増加させることになり，正常な実行を偽装した攻撃が容易に行える．

一方，Feng らの手法 [4] や安部らの手法 [11] 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プログラムの実行状態をより特定できるようにしている．Feng らの手法を例に説明し，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．このコールスタックから収集した戻りアドレスのリストを Virtual Stack List と呼ぶ．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．具体的には，n 番目のシステムコールが呼び出されたときと，n+1 番目のシステムコールが呼び出されたときの Virtual Stack List を比較し，共通し

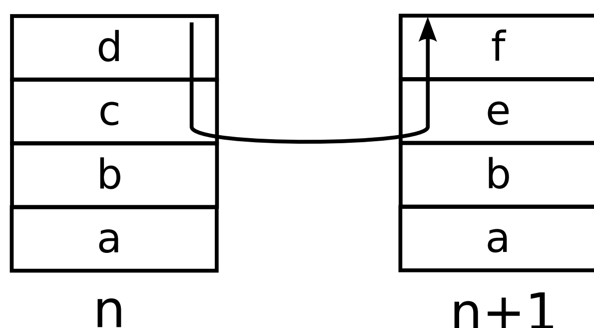


図 2.1: Virtual Path

ている戻りアドレスを確認する．図 2.1 のように，n と n+1 の Virtual Stack List で戻りアドレス a，b が共通していた場合，d，c の戻りアドレスを積んだ関数が終了した後，e，f の戻りアドレスを積んだ関数が呼び出されたことが分かる．この間数の終了，呼び出しの遷移 (図 2.1 の矢印) を Virtual Path と呼ぶ．

Feng らの手法ではトレーニングモードで対象のプログラムを実行したときの Virtual Path を収集する．そして，検知モードで実行したときの Virtual Path が，トレーニングモードで収集された中にあるかを調べることで対象のプログラムの動作を確認する．安部らの手法は Feng らの Virtual Path を元に，監視対象のプログラムのソースコードを静的解析を行い作成した正常な動作規則を用いてプログラムの監視を行う．

Feng らと安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．この問題点について，安部らの手法を例に述べる．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 についての動作規則が図 2.2 のように表されていたときを考える．図 2.2 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 が終了した後に関数 B が呼び出されたと考えることができ，正常な動

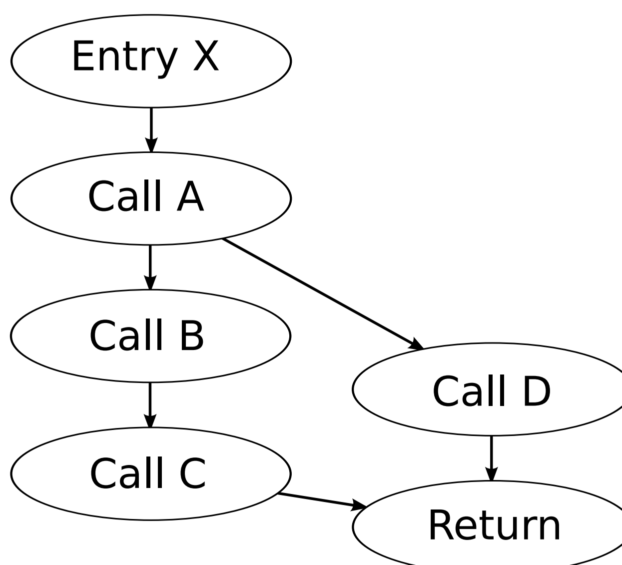


図 2.2: 関連研究の正常な動作規則例

作規則とも一致する．

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，安部らの手法では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，安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 2.3 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 2.3 の遷移 ②) となり，動作規則に定義され

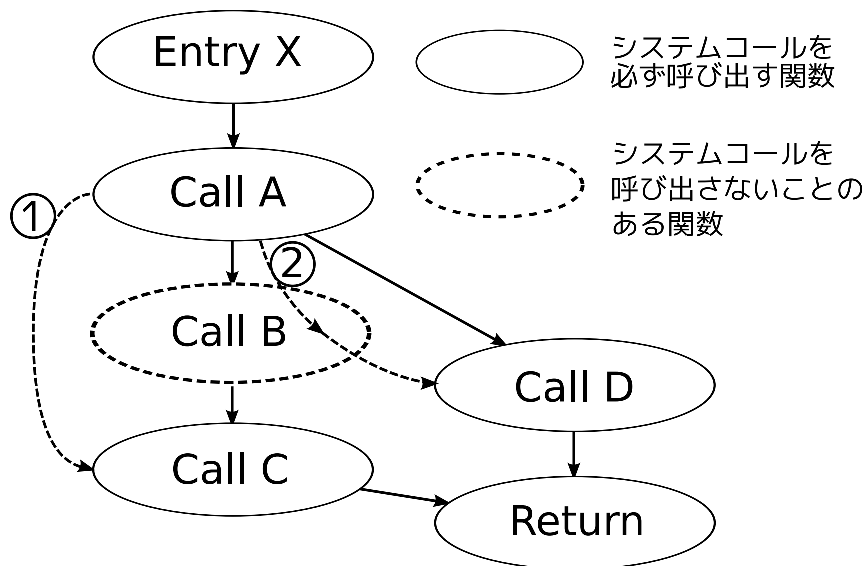


図 2.3: 関連研究の正常な動作規則の問題点

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，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．e-NeXSh ではフックするライブラリ関数が少ないためプログラムの実行状態の把握が不十分になり，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．

第3章

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 ライブラリ関数を呼び出したときに情報を取得

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 ライブラリ関数の呼び出し確認

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，攻撃者が libelem を回避してライブラリ関数を呼び出したり，攻撃コードから直接システムコールを発行したとしても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フック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．提案手法ではプログラムの実行をより詳細に把握するために，e-NeXSh より多くのライブラリ関数の呼び出しをフックする．そのため，ライブラリ関数をフックする度にシステムコールを呼び出すと，e-NeXSh に比べてオーバヘッドの増加がより顕著になってしまう．そこで，提案手法ではそのシステムコールを省き，ライブラリ関数がフックされたか否かを表す情報を，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．

コールスタック履歴には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も書き込む．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し，ライブラリ関数がフックされていたか，動作規則に従った順で呼び出されていたかを確認する．

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．

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．

第4章

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 4.1 を用いて述べる．図 4.1 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および，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).

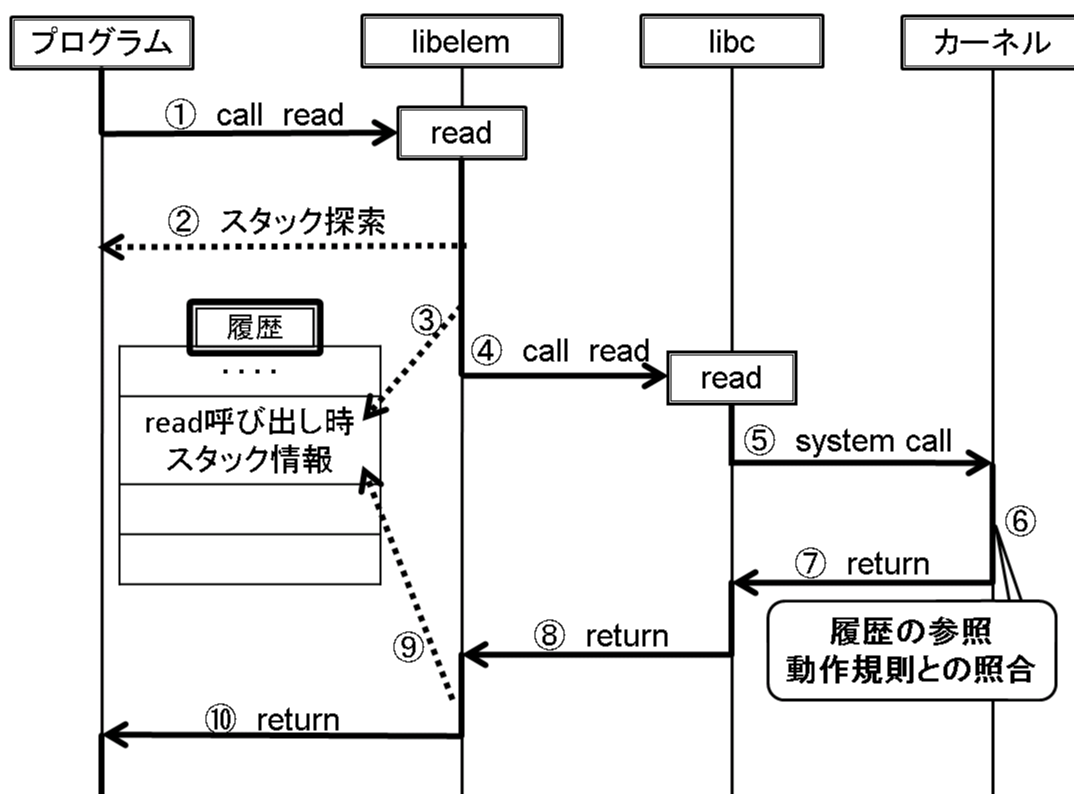


図 4.1: 提案システムの全体像

4.2 動作規則

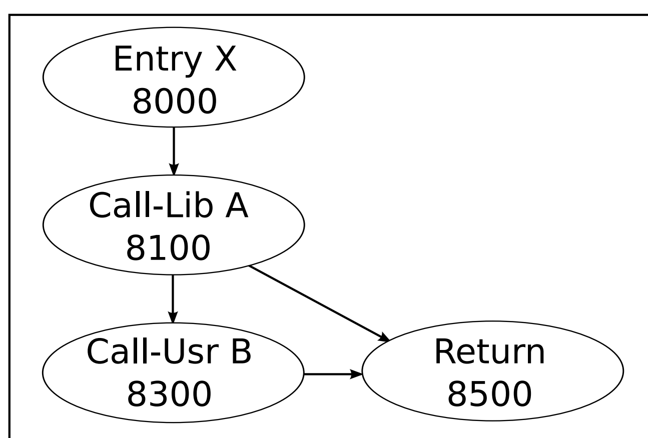
動作規則の作成手法とフォーマットについて述べる．

4.2.1 動作規則の作成例

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 4.2 に示す．監視対象の実行ファイルを逆アセンブルした結果，図 4.2(a) のようなアセンブリコードが得られたとき，図 4.2(b) のような動作規則を作成する．動作規則は 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

```
8000  X:
 ...
8100  call A
 ...
8200  je 8400
 ...
8300  call B
 ...
8500  return
```

(a) assembly code



(b) 動作規則

図 4.2: 動作規則の作成

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．そして，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一緒に記録する．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．

```
#define RULE_INDU_MN "INDU"       /* Magic Number */
#define RULE_INDU_NM "UDNI"
struct ind_h{
  char mn[4] = RULE_INDU_MN;
  unsigned int num;               /* ユーザ関数の数 */
  struct ind_usrf* usrf;
  char nm[4] = RULE_INDU_NM;
};

struct ind_usrf{
  unsigned int addr;              /* 関数の先頭アドレス */
  int off;                        /* 定義位置までのオフセット */
  struct ind_usrf* usrf;
};
```

図 4.3: Index のフォーマット

4.2.2 動作規則のフォーマット

動作規則はIndex と Function block で構成され，Function block は複数の Rule block によって構成されている．

Index

Index のフォーマットを図 4.3 に示す．Index は，ヘッダである構造体 ind_h とユーザ関数ごとに作成する複数の構造体 ind_usrf で構成される．

構造体 ind_h は，マジックナンバー (mn)，定義するユーザ関数の数 (num)，構造体 ind_usrf へのポインタ (usrf) で構成されている．構造体 ind_usrf は定義しているユーザ関数の先頭アドレス (addr) と，対応する Function block の定義への相対アドレス (off)，及び構造体 ind_usrf へのポインタ (usrf) で構成される．構造体 ind_usrf はユーザ関数の先頭アドレスが小さい順に，プログラムに定義されているユーザ関数の数だけ連なる．

```
#define RULE_UFB_MN "UDEF"      /* Magic Number */
#define RULE_UFB_NM "FEDU"      /* Magic Number */
struct ufb_h{
  char mn[4] = RULE_UFB_MN;
  unsigned int num;             /* rule block の数 */
  struct ufb_rb* rb;
  char nm[4] = RULE_UFB_NM;
};
```

図 4.4: Function block のフォーマット

Function block

Function block のフォーマットを図 4.4 に示す．Function block は，構造体 ufb_h は構造体 ind_usrf と 1 対 1 対応している．構造体 ufb_h はマジックナンバー (mn)，定義する Rule block の数 (num)，Rule block の構造体 ufb_rb へのポインタ (rb) で構成される．

Rule block

Rule block のフォーマットを図 4.5 に示す．Rule block はノードの情報をもつ構造体 ufb_rb と，遷移可能ノードの情報をもつ構造体 ufb_rb_nn で構成されている．構造体 ufb_rb はノードのタイプ (ct)，遷移可能ノードの数 (nn)，呼び出し元アドレス (ea)，呼び出し先アドレス (ca)，及び遷移先ノードの情報を持つ構造体 ufb_rb_nx へのポインタ (nx) で構成されている．構造体 ufb_rb_nn は遷移先ノードの Rule block の定義までの相対アドレス (off)，及び構造体 ufb_rb_nn へのポインタ (nx) で構成され，構造体 ufb_rb に定義されている遷移可能ノードの数 (nn) だけ連なる．

```
struct ufb_rb{
  unsigned short int ct;        /* Call Type (RULE_RB_CT*) */
  unsigned short int nn;        /* Next Num */
  unsigned int ea;              /* Executing Address */
  unsigned int ca;              /* Call Address */
  struct ufb_rb_nx* nx;
};

struct ufb_rb_nx{
  int off;
  struct ufb_rb_nx* nx;
};

/*--- Call Type (Option) ---*/
#define RULE_RB_CT_HD 0         /* Function Head (Function Entry) */
#define RULE_RB_CT_SC 1         /* System Call */
#define RULE_RB_CT_LF 2         /* Libraly Function Call */
#define RULE_RB_CT_UF 4         /* User Function Call */
#define RULE_RB_CT_FP 5         /* ポインタを用いた関数呼び出し */
#define RULE_RB_CT_LJ 6         /* longjmp */
#define RULE_RB_CT_PJ 7         /* ポインタを用いた Jump */
#define RULE_RB_CT_RT 9         /* Return */
```

図 4.5: Rule block のフォーマット

4.2.3 動作規則のサイズ

作成した動作規則のサイズを表 4.1 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予定である．

表 4.1: 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

4.2.4 動作規則の管理

作成した動作規則はGNUのobjcopyユーティリティによりリンク可能なオブジェクトに変換する．そして，監視対象プログラムにリンクすることで，実行時に動作規則がメモリ上に展開される．

4.3 ライブラリ関数の呼び出し確認

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．環境変数LD_PRELOADに指定したライブラリは，ライブラリ検索順序の中でもっとも早く検索される．プログラム実行時には，プログラムから呼び出されるライブラリ関数がどのライブラリに定義されているかが，ライブラリ検索順序にしたがって検索される．そして，複数のライブラリで同じ関数を定義していた場合，先に検索されたライブラリに定義されているライブラリ関数が呼び出されることになる．

そこで，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を指定した．そのため，監視対象からライブラリ関数を呼び出した場合，libelemのほうがlibcよりライブラリ検索順序が早くなる．そして，本来呼び出されるはずであったlibcの関数を呼び出す代わりに，libelemに定義された同型の関数を呼び出させることができる．

LD_PRELOADに指定したlibelemによりライブラリ関数をフックした後，監視対象のプログラムの動作を変えないために，本来呼び出されるべきであったlibcに定義されている同名の関数を呼び出す必要がある．そのためlibelemで定義している関数

では，dlsym 関数を用いて本来呼び出されるべきであった関数のアドレスを入手する．dlsym 関数を用いると，ライブラリ検索順序に従い，現在のライブラリ (libelem) 以降のライブラリで，任意の関数が最初に定義されているアドレスを探すことができる．ライブラリ検索順序では libelem の次は libc になっているため，libc での関数の定義アドレスを取得することができる．

4.4　コールスタック履歴の作成

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発行しなかったとしても，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．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．

コールスタック履歴はシステムコール発行時にカーネルから参照される．libc 関数をフックした時にコールスタック履歴に書き込んだ記録は，次にカーネルから参照されるまで保持する必要がある．システムコールを発行しなかった libc 関数が連続すると，コールスタック履歴にはカーネルから未参照の記録がたまっていく．そのため，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た．

4.5 動作規則との照合

カーネルでは，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発行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ていく．そして，現在の実行状態に一致する動作規則上のノードまでの遷移を確認する．戻りアドレスリストと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 4.6 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 4.6 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

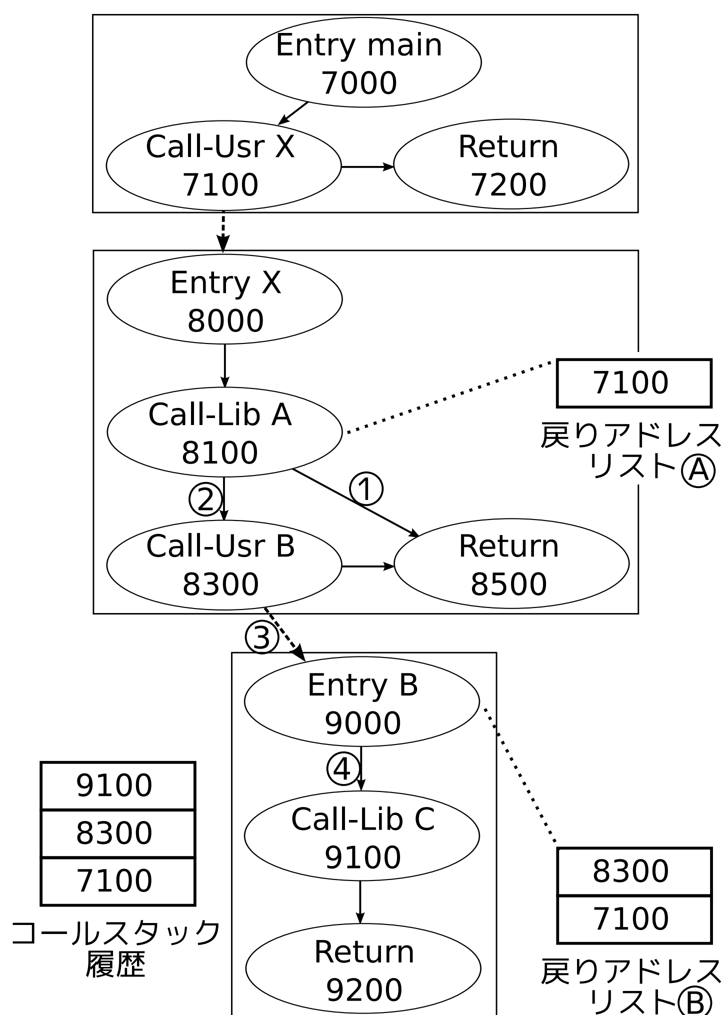


図 4.6: 動作規則との照合

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，監視を開始する．また，監視対象プログラムが fork と 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．監視システムではシステムコールを発行したプロセスの pid が，監視対象の pid リストに含まれているか否かの判定を行っている．そのため，fork および clone システムコールの戻り値を取得し，監視対象 pid リストに追加する．

第5章

評価

前章で実装したシステムの評価を行った．評価には，監視オーバヘッドの測定，branching factor の計算および単純な攻撃コードによる検証を行った．評価環境は表 5.1 に示す．

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 5.2 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．

inetd の測定方法について図 5.1 を用いて説明する．マシン 2 で動作しているクライアントからマシン１に接続する．マシン１で動作している inetd は select によりポートへの接続を待っている．クライアントからの接続があると fork を行い，親プロセスはふたたび select を行い，子プロセスは接続されたポートに対応したサーバを起動する．inetd より起動されたサーバはクライアントにメッセージを送り終了する．このサーバが終了すると親プロセスである inetd に SIGCHILD シグナルが送信され，対応したシグナルハンドラが実行される．メッセージを受け取ったクライアントはソケットをクローズし，通信を終了する．

表 5.1: 評価環境

CPU	Pentium4 3.2GHz(HT 無効)
MEM	1GB
linux kernel のバージョン	2.6.17.8
libc のバージョン	2.4

表 5.2: 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	1.69(1.00)	2.26(1.34)	3.33(1.97)
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

inetd の実行時間のとして，クライアントがマシン１に接続してからソケットをクローズするまでの動作を 1000 回繰り返したときの時間を測定した．監視対象としたのは inetd のみであり，クライアントや inetd から起動されるサーバは監視対象ではない．また，表 5.2 の inetd は (マシン 1)≠(マシン 2) の場合であり，inetd(local) は (マシン 1)=(マシン 2) の場合である．

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．

wc のように，システムコールの数に対してフックを行ったライブラリ関数の数が非常に多い場合は，監視オーバヘッドが大きくなってしまう．しかし，inetd のようにシステムコールの数とフックを行ったライブラリ関数の数が同じくらいであれば，監視

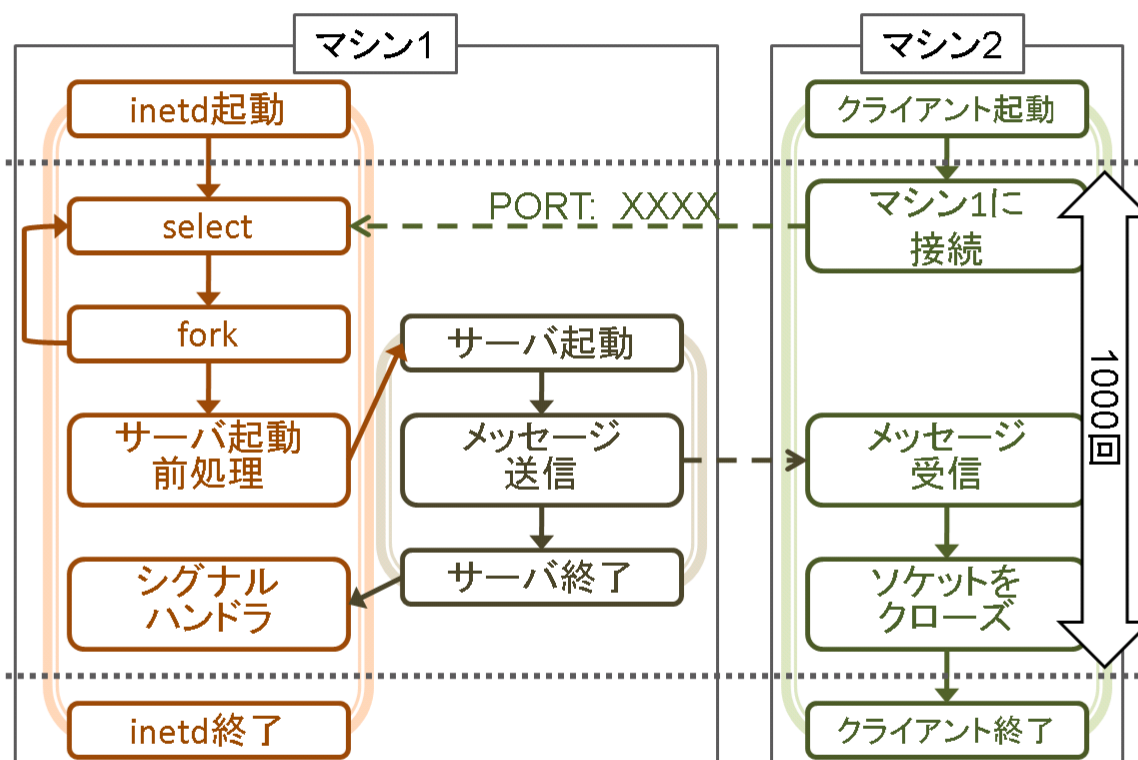


図 5.1: inetd の監視オーバヘッド測定

オーバヘッドは許容範囲内である．

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[7][9] 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．

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

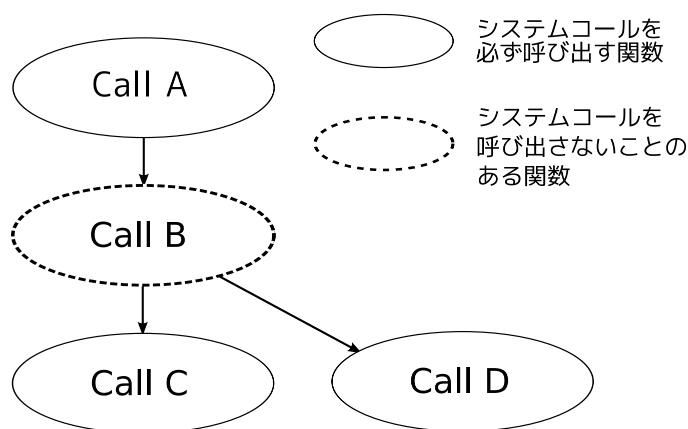


図 5.2: ライブラリ関数呼び出しが必ず把握できるとき (提案手法) の動作規則

表 5.3: 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

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．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群から，必ずシステムコールを発行するライブラリ関数群を除いたものが，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群である．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は，呼び出しが把握できない可能性があることを考慮した遷移を動作規則に追加する．

提案手法での動作規則が図 5.2 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 5.3 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 5.3 に，把

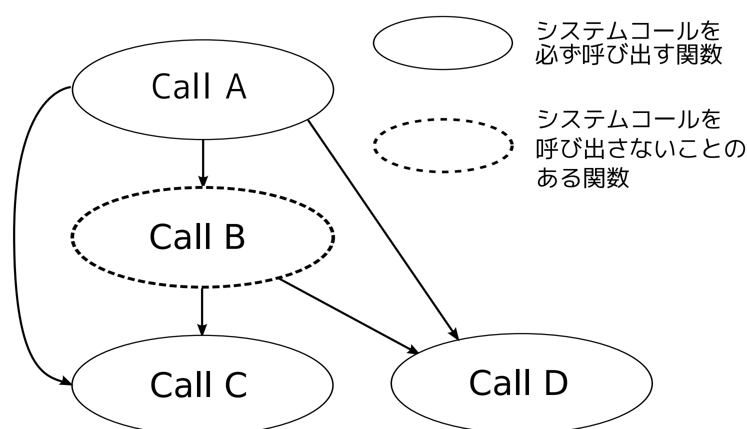


図 5.3: ライブラリ関数呼び出しが把握できるとは限らないときの動作規則

表 5.4: 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

握できるとは限らない場合の branching factor を表 5.4 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．そのため，inetd のほうが提案手法により検知精度が大きく向上することが期待できる．

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，提案システムで攻撃を防止できるか検証した．

```
#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 5.4: サンプルプログラム 1

5.3.1 サンプルプログラムの説明

まずは，文献 [5] を参考にして作成した，libc ベース攻撃のサンプルプログラム 1(図 5.4) の説明をする．関数 main から関数 func が呼び出された直後のスタックは図 5.5(a) のように main への戻りアドレス，退避済みベースポインタ，関数 func の引数が積まれている．

関数 func の最初の行

```
void **fp = (void **) &fp;
```

によって変数 fp は fp 自身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 func の戻りアドレスを指すようなる．fp[2] にライブラリ関数 execve のアドレスを代入することで，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ることができる．同様に，fp[4]~fp[6] にライブラリ関数

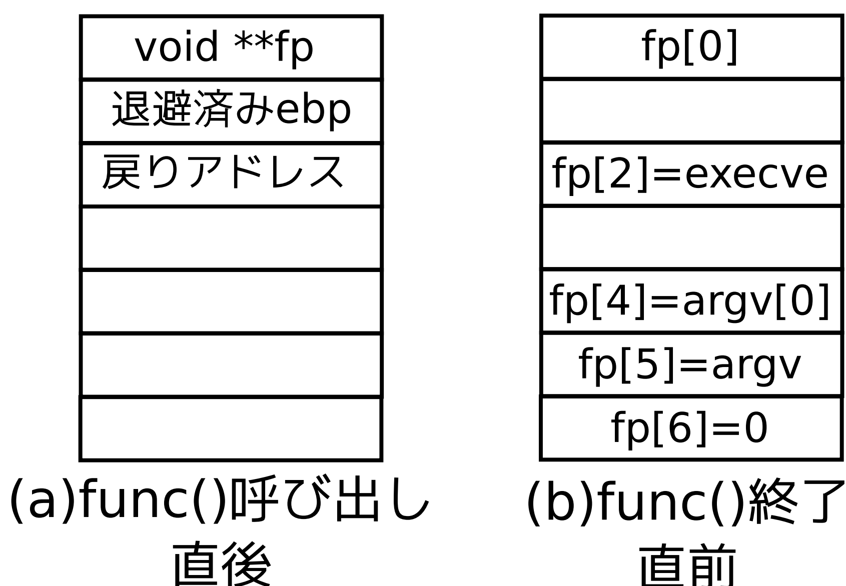


図 5.5: サンプルプログラム実行時のスタック

execve の引数を代入している．なお，ライブラリ関数 execve の定義は次のとおりである．

```
int execve(const char *filename, char *const argv [], char *const envp[]);
```

関数 func 終了直前のスタックは 5.5(b) のようになっており，戻りアドレスが書き換えられているため return 命令により不正に execve に実行が移される．fp[4]~fp[6] に代入した値が引数として渡されるため，/bin/sh が実行される．

5.3.2 フックしないライブラリ関数を利用した攻撃

libelem にライブラリ関数 execve を定義していない場合，ライブラリ関数 execve の呼び出しがフックされない．この場合において，提案システム上でサンプルプログラム 1(図 5.4) を実行した．

サンプルプログラム 1 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．提案システムではこのシステムコールをフックしたとき，コールスタック履歴を確認する．しかし，コールスタック履歴にはライ

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残っていない．提案システムは libelem によりフックされていないライブラリ関数からのシステムコール発行を認めない．そのため，ライブラリ関数 execve から呼び出された execve システムコールを処理せず，サンプルプログラム 1 の実行を強制的に終了させた．

5.3.3 フックするライブラリ関数を利用した攻撃

libelem にライブラリ関数 execve を定義しておき，libelem によりライブラリ関数 execve の呼び出しがフックされるようにした．そして，提案システム上でサンプルプログラム 1(図 5.4) を実行した．

サンプルプログラム 1 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．提案システムではこのシステムコールをフックしたとき，コールスタック履歴にはライブラリ関数 execve が libelem によりフックされて実行中であるという記録が残っている．そのため，提案システムではコールスタック履歴を読み込むが，そこに書かれているライブラリ関数 execve の戻りアドレスがユーザ関数内のアドレスではなかった．ゆえに，提案システムはライブラリ関数 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，サンプルプログラム 1 の実行を強制的に終了させた．

```
#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 5.6: サンプルプログラム 2

5.3.4　フックするライブラリ関数を利用し，スタックも偽装した攻撃

libelem によりライブラリ関数 execve の呼び出しがフックされるように設定し，さらにライブラリ関数 execve の戻りアドレスをユーザ関数内のアドレスに設定したサンプルプログラム 2(図 5.6) を提案システム上で実行した．サンプルプログラム 1 からの変更点は関数 func に次の 1 行を追加し，ライブラリ関数 execve の戻りアドレスをユーザ関数 func のアドレスに書き換えたことである．

```
fp[3] = func;
```

これにより，ライブラリ関数 execve が呼び出された直後のスタックは図 5.7 のようになる．

サンプルプログラム 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．提案システムではこのシステムコールをフッ

execveの
局所変数
退避済みebp
fp[3]=func
fp[4]=argv[0]
fp[5]=argv
fp[6]=0

図 5.7: execve() 呼び出し直後のスタック

クしたとき，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残っている．また，コールスタック履歴に書かれているライブラリ関数 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と照合する．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，提案システムは execve システムコールを処理せず，サンプルプログラム 2 の実行を強制的に終了させた．

以上の検証により，提案システムで実際に攻撃を検知できることを確認できた．

第6章

考察

提案システムで検知可能な攻撃，提案システムに対する攻撃，提案システムの問題点について考察を行った．

6.1　検知性能の考察

文献 [8] で紹介されている 20 種のバッファオーバフロー攻撃パターンが，提案により検知可能か否かの考察を行った．そして，20 種のバッファオーバフロー攻撃パターンを既存のセキュリティシステムに適応した結果との比較を行った．

6.1.1　バッファオーバフロー攻撃を行う 3 つの要素

文献 [8] では，バッファオーバフロー攻撃を 3 つの要素により分類している．それは，バッファオーバフローテクニック，バッファオーバフローを発生させる場所，攻撃対象の 3 つである．その 3 つの要素について説明する．

バッファオーバフローテクニック

バッファオーバフローを用いた攻撃は，直接攻撃対象を書き換える攻撃とポインタの参照先を攻撃対象に書き換える攻撃に分けられる．直接攻撃対象を書き換える攻撃は，バッファオーバフローを発生させる起点から攻撃対象までを書き換える．そのため攻撃対象のデータがバッファオーバフローを発生させる起点より後 (スタックの底の

```
static int global_const=1;              // data セグメント
static int global_var;                  // bss セグメント

int main(void){
        int local_dynamic_var;          // スタック
        static int local_static_const=1; //data セグメント
        static int local_static_var     //bss セグメント
        int *buf_ptr=(int *)malloc(32)  // ヒープ
        …
}
```

図 6.1: 変数の確保とデータの置場

方) にある必要がある．一方，ポインタの参照先を攻撃対象に書き換える攻撃は，バッファオーバフローを発生させてポインタの参照先を攻撃対象に付け変えることで，ポインタを使って攻撃対象を書き換える．そのためポインタがバッファオーバフローを発生させる起点より後にある必要はあるが，攻撃対象はメモリ上のどこにあってもよい．

バッファオーバフローを発生させる場所

オーバフローを発生させる場所は，スタックセグメントもしくはヒープ/bss/data セグメントに分けられる．スタックセグメントとはプログラムスタックを配置するセグメントで，実行中の関数の戻りアドレス，引数，ローカル変数などが置かれる．スタックセグメントでバッファオーバフローを発生させると，直接戻りアドレスや退避済みベースポインタを書き換えることができる．

ヒープは関数 malloc で確保されるメモリ領域である．bss セグメントは静的変数や大域変数のうち，初期値が実行ファイルで定義されているデータを置く場所である．data セグメントは静的変数や大域変数のうち，初期化されていないデータを置く場所である．

変数の定義の仕方とデータの置かれる場所の対応を図 6.1 に示す．変数の置かれる場所はコメントとして書かれている．

攻撃対象

ここで扱う攻撃対象は，書き換えることでプログラムの実行フローを変えることができるデータである．攻撃対象は戻りアドレス，退避済みベースポインタ，関数ポインタ，longjmp のバッファに分けられる．また，関数ポインタと longjmp のバッファはそのデータの置かれる領域により分けて考える必要がある

戻りアドレス，退避済みベースポインタの置き場所は共にスタックである．一方関数ポインタと longjmp のバッファは，スタックに置かれた場合とヒープ/bss/data セグメントに置かれた場合とが考えられる．さらに，関数ポインタと longjmp のバッファはスタックに置かれた場合でも，その値が変数としてあたえられた場合と，関数の引数としてあたえられた場合が考えられる．

攻撃対象が変数であるか，関数の引数であるかは，検知の可/不可に大きく関わってくる場合がある．例えば攻撃対象が，6.1.4 で説明する既存手法の StackGuard 及び ProPolice が挿入する「カナリア」値より前にスタックに積まれるか，後に積まれるかということが検知に影響してくる．攻撃対象が変数であった場合，攻撃対象は「カナリア」より後にスタックに積まれるが，攻撃対象が関数の引数であった場合は「カナリア」より前にスタックにつまれることになる．

なお，関数ポインタ func_ptr が次ののように定義されていた場合，func_ptr には引数が char 型で戻り値が int 型の関数の実行ポイントを代入することができる．

```
int (*func_ptr) (char);
```

そのため攻撃者が関数ポインタを書き換えることができれば，プログラムに任意の関数の呼び出させることができる．また，longjmp は setjmp とセットで使用される．setjmp で実行環境 (プログラム中の実行位置やスタック等) をバッファに保存し，longjmp によって復元する．このバッファを攻撃者が書き換えることができれば，longjmp により復元される実行環境を任意に変更することができる．

6.1.2　20種のバッファオーバフロー攻撃パターン

文献 [8] にはバッファオーバフローテクニック，オーバフローを発生させる場所，攻撃対象の 3 つの要素の可能な組合せで 20 種類のバッファーオーバフロー攻撃パターンが定義されている．20 種の攻撃パターンを以下にまとめる．

(a). スタックセグメントでバッファオーバフローを発生させ，以下の攻撃対象を直接書き換える

1. 戻りアドレス

2. 退避済みベースポインタ

3. 変数の関数ポインタ

4. 関数引数の関数ポインタ

5. 変数の longjmp のバッファ

6. 関数引数の longjmp のバッファ

(b). ヒープ/bss/data セグメントでバッファオーバフローを発生させ，以下の攻撃対象を直接書き換える

7. 関数ポインタ

8. longjmp のバッファ

(c). スタックセグメントでバッファオーバフローを発生させ，ポインタの参照先を以下の攻撃対象に書き換える

9. 戻りアドレス

10. 退避済みベースポインタ

11. 変数の関数ポインタ

12. 関数引数の関数ポインタ

13. 変数の longjmp のバッファ

14. 関数引数の longjmp のバッファ

(d). ヒープ/bss/data セグメントでバッファオーバフローを発生させ，ポインタの参照先を以下の攻撃対象に書き換える

15. 戻りアドレス

16. 退避済みベースポインタ

17. 変数の関数ポインタ

18. 関数引数の関数ポインタ

19. 変数の longjmp のバッファ

20. 関数引数の longjmp のバッファ

6.1.3 提案システム上で攻撃パターンを再現したと仮定したときの評価

提案システム上で攻撃パターンを再現したと仮定したときの評価を行った．提案システムではオーバフロー攻撃自体を検知するのではなく，オーバフロー攻撃によって引き起こされる実行の変化を見付けることができる．

しかし，提案システムでは動作規則に表せない実行の変化は見付けることができない．現在の実装では，動作規則には関数ポインタや longjmp による実行の変化を表すことができないため，関数ポインタや longjmp を利用した攻撃は提案システムでは検知できない．関数ポインタあるいは longjmp により，どのように実行が変化する可能性があるかを動作規則作成時に解析し，動作規則に反映することができれば提案システムにより防ぐこともできると考えている．

6.1.4 既存のシステムとの比較

20 種のオーバフロー攻撃パターンを用いて，提案システムと及び既存手法の比較を行った．比較を行った既存手法は StackGuard，Stack Shield，ProPolice，Libsafe である．既存手法の評価結果は文献 [8] を参照した．各既存手法についての説明と，評価結果についてまとめる．

StackGuard

StackGuard[2] では，コンパイラにより関数を呼び出したときにスタックの戻りアドレス直前に「カナリア」呼ばれる値を挿入するように変更する．そして，関数が終了して戻りアドレスにジャンプする前に,「カナリア」の値を確認する.「カナリア」値が変更されていればプログラムを終了させる．

攻撃者が戻りアドレスを直接書き換えるためには，スタック上でバッファオーバフローを発生させる起点となる変数から，戻りアドレスまでのスタックをすべて書き換える必要がある．そのため，この手法により戻りアドレスを書き換えるときには，その直前にある「カナリア」も共に書き換えられるため，StackGuard で検知することができる．

Stack Shield

Stack Shield[6] は，関数処理の開始時に戻りアドレスを通常のスタックとは別の安全な場所に保存する．そして，関数が終了する直前に，戻りアドレスをスタック上に復帰させる．戻りアドレスが通常のスタック上に存在しないため，攻撃者はスタック上でバッファオーバフローを発生させても，戻りアドレス書き換えることができなくなる．

ProPolice

ProPolice[10] は StackGuard を元に,「カナリア」の値を使用しているが次の点の改良を加えている．文字列配列のローカル変数は他のローカル変数よりスタックの底の方に配置されるようにし，退避済みベースポインタの直前 (文字列のローカル変数と退避済みベースポインタの間) に「カナリア」値を挿入する．

これにより文字列配列のローカル変数がオーバフローしても，ポインタなどの他のローカル変数に影響をあたえることがない．さらに，退避済みベースポインタ及び戻りアドレスの書き換えは「カナリア」により防ぐことができる．

表 6.1: スタックセグメントでバッファオーバフローを発生させ，攻撃対象を直接書き換える

検知手法＼攻撃対象	戻りアドレス	退避済みベースポインタ	関数ポインタ		longjmp のバッファ	
			変数	関数引数	変数	関数引数
提案システム	○	○	×	×	×	×
StackGuard	○	○	×	×	×	×
Stack Shield	○	○	×	×	×	×
ProPolice	○	○	○	×	○	×
Libsafe	○	○	×	○	×	○

表 6.2: ヒープ/bss/data セグメントでバッファオーバフローを発生させ，攻撃対象を直接書き換える

検知手法＼攻撃対象	関数ポインタ	longjmp のバッファ
提案システム	×	×
StackGuard	×	×
Stack Shield	×	×
ProPolice	×	×
Libsafe	×	×

Libsafe

Libsafe[1] は strcpy や strcat 等の脆弱性のあるライブラリ関数用のラッパー関数を提供する．ラッパー関数では，関数の引数に確保されているサイズを計算し，その境界をチェックする．これにより，ライブラリ関数を適切に使用していなかったとしても，オーバフローの発生を防ぎ，安全に実行することができる．

評価結果の比較

評価結果を表 6.1，6.2，6.3，6.4 にまとめる．なお，提案システムの評価は，提案システム上で攻撃パターンを再現したと仮定したときの評価であるが，既存システムの評価は実際に既存システム上で攻撃を行ったときの評価である．

表 6.3: スタックセグメントでバッファオーバフローを発生させ，ポインタの参照先を攻撃対象に書き換える

攻撃対象 / 検知手法	戻りアドレス	退避済みベースポインタ	関数ポインタ		longjmp のバッファ	
			変数	関数引数	変数	関数引数
提案システム	○	○	×	×	×	×
StackGuard	×	○	×	×	×	×
Stack Shield	○	○	×	×	×	×
ProPolice	○	○	○	○	○	○
Libsafe	×	×	×	×	×	×

表 6.4: ヒープ/bss/data セグメントでバッファオーバフローを発生させ，ポインタの参照先を攻撃対象に書き換える

攻撃対象 / 検知手法	戻りアドレス	退避済みベースポインタ	関数ポインタ		longjmp のバッファ	
			変数	関数引数	変数	関数引数
提案システム	○	○	×	×	×	×
StackGuard	×	×	×	×	×	×
Stack Shield	○	×	×	×	×	×
ProPolice	×	×	×	×	×	×
Libsafe	×	×	×	×	×	×

評価結果から，既存のシステムでは，スタックセグメントでのバッファオーバフロー (表 6.1，6.3) は検知できる場合が多いが，ヒープ/bss/data セグメント上でのバッファオーバフロー (表 6.2，6.4) は検知できない場合が多いことが分かる．既存の手法ではバッファオーバフロー自体を検知することが目的であるため，テクニックや発生場所に検知の可/不可が依存する．

一方，提案システムではバッファオーバフロー攻撃によって引き起こされる実行の変化を見つけるため，バッファオーバフローを発生させるテクニックや場所に依存しない．しかし，提案システムでは関数ポインタまたは longjmp のバッファが攻撃対象のとき，攻撃を検知するとができない．今後，関数ポインタで呼び出される関数と longjmp によるジャンプ先の候補を動作規則の作成段階で特定し，検知に反映することにより

すべての場合で検知できるようになる．

6.2　検知可能な攻撃

攻撃者が提案システムにより守られているプログラムを攻撃した場合について考える．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．

6.3　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すことはできない．

提案システムで動作規則は，プログラムが実行されたときにユーザ空間にマッピングされるように設定している．そのためプログラムから動作規則にアクセスすることが可能である．そこで攻撃者は動作規則を書き換えることで，任意の攻撃が提案システムで許可されるようにすることが考えられる．現時点では未実装であるが，動作規則をユーザ空間にマッピングされてからすぐ (main が実行される前) に動作規則のアクセス権を設定しなおし，書き換えが行えないようにする対策が考えられる．または，プログラムのヘッダ情報を書き換え，動作規則を書き換え不可に設定されたメモリにロードする対策も考えられる．

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かもしれない．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．安全性を高めるためにコールスタック履歴をカーネル空間に移すこす方法も考えられる．具体的には，libelem でライブラリ関数をフックして確認した実行状態を，システムコールによりカーネルに知らせ，カーネル空間のメモリに記録する．カーネル空間に記録しておくことで，悪意のあるユーザによって不正に書き換えられることがなくなるが，システムコール発行によるオーバヘッドが発生する．

libelem でライブラリ関数をフックするごとにシステムコールを発行し，コールスタック履歴をカーネル空間に記録することによる，wc プログラムの監視オーバヘッド

表 6.5: wc 監視においてコールスタック履歴の記録する空間によるオーバヘッド比較

コールスタック履歴の記録場所	通常実行	動作規則と照合
ユーザ空間	1.69(1.00)	3.33(1.97)
カーネル空間	1.69(1.00)	11.24(6.65)

測定結果を表 6.5 に示す．

libelem によるライブラリ関数のフックは 1500 万回フックしたため，カーネル空間にコールスタックを記録したほうがシステムコールを 1500 万回多く発行したことになる．カーネル空間にコールスタック履歴を記録した場合ではオーバヘッドが非常に大きいため，ユーザ空間にコールスタック履歴を記録し，コールスタック履歴の安全性を向上させていきたいと考えている．

6.4 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．そのため，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，すべての関数を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．また，branching factor を求めるときにも関数ポインタの存在を考慮していない．通常の関数呼び出しを定義したノードから呼び出される関数の候補は 1 つだけだが，関数ポインタを用いた関数呼び出しを定義したノードから呼び出される関数の候補は複数存在する．関数ポインタで呼び出される関数をより限定できる動作規則ほど，branching factor によりよい評価値が得られることが望ましい．

また本システムでは，監視対象プログラムが実行される前に libc あるいは libc に定義されている関数が改変されていたとしても，その改変を検知することができない．さらに，改変された libc を悪用した攻撃も防ぐことができない．この問題の解決するに

は，監視対象プログラムの実行前あるいは実行中に，libc が改変されていないことを確認する機構を組み込む必要がある．例えば，libc に定義されている関数においてもユーザ関数と同様の動作規則を作成し，システムコールが発行されたときのコールスタックを用いてライブラリ関数の実行状態を監視することも考えられる．しかしこの場合，さらなるオーバヘッドの増加と動作規則の肥大化が懸念される．

第7章

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．

謝辞

本研究のために，多大な御尽力を頂き，御指導を賜った名古屋工業大学の松尾啓志教授，津邑公暁助准教授，齋藤彰一准教授，松井俊浩助教に深く感謝致します。

また，本研究の際に多くの助言，協力をして頂いた松尾・津邑研究室，齋藤研究室の方々に深く感謝致します。

参考文献

[1] A.Baratloo, T.Tsai, and N.Shingh. Libsafe: Protecting critical elements of stack. In *White Paper http://www.research.avayalabs.com/project/libsafe/*, December 1999.

[2] C.Cowan, C.Pu, D.Maier, J.Walpole, P.Bakke, S.Beattie, A.Grier, P.Wagle, Q.Zhang, and H.Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attack. In *Proceedings of the 7th USENIX Security Conference*, pp. 63–78, January 1998.

[3] C.Warrender, S.Forrest, B.Pearlmutter, and B.Pearlmutter. Detecting intrutions using system call: Alternative data models. In *Proc. 2001 IEEE Symposium on Security and Privacy*, pp. 156–168. Oakland, 2001.

[4] H.H.Feng, O.M.Kolesnikov, P.Fogla, W.Lee, and W.Gong. Anomaly detection using call stack information. IEEE Symposium on Security and Privacy，Berkeley，CA, pp. 62–77, 2003.

[5] Gaurav S.Kc and Angelos D.Keromytis. e-nexsh: Achieving an effectively non-executable stack and heap via system-call policing. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*. Tucson，AZ, December 2005.

[6] Vendicator. Stack shield technical info v0.7. http://www.angelfire.com/sk/stackshield/. January 2001.

[7] David Wagner and Drew Dean. Interusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, pp. 144–155, 2001.

[8] John Wilander and Marian Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium (NDSS)*, pp. 123–130, February 2003.

[9] 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.

[10] 江藤博明, 依田邦和. Propolice: スタックスマッシング攻撃検出手法の改良. コンピュータセキュリティ研究会, pp. 181–188, July 2001.

[11] 安部洋丈, 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 Vol. 45, No. SIG 3(ACS 5), pp.11-20, 2004.3.