情報処理学会論文誌　コンピューティングシステム　Vol. 2　No. 4　48–63 (Dec. 2009)

# 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[⋆1]

嶋村　誠[†1]　河野健二[†1,†2]

遠隔から攻撃コードをネットワークを介して挿入するリモートコードインジェクション攻撃はセキュリティ上の大きな問題の 1 つである．これに対し，攻撃コードを検知・解析するシステムとして，ネットワーク・コードエミュレータが提案されている．ネットワーク・コードエミュレータでは攻撃コードの疑似実行を行うことにより，攻撃コードを精度良く検知したり，攻撃コードの振舞いを詳細に解析したりできる．また，攻撃コードを実行して解析を行うため，暗号化や難読化を施された攻撃コードにも耐性がある．本論文では，被害プロセスのメモリ上のデータを攻撃コードの一部として利用するメモリスキャン攻撃を用いると既存のネットワーク・コードエミュレータによる解析を妨害できることを示し，メモリスキャン攻撃も解析できるネットワーク・コードエミュレータである Yataglass+を提案する．実際に Yataglass+のプロトタイプを作成し，実際の攻撃コードにメモリスキャン攻撃を適用し実験を行った結果，Yataglass+は正しくメモリスキャン攻撃を適用した攻撃コードを解析できた．

# Yataglass+: Network-level Behavior Analysis of Shellcode That Incorporates Memory-scanning Attacks

Makoto Shimamura[†1] and Kenji Kono[†1,†2]

Remote code-injection attacks are one of the most serious problems in computer security because they allow attackers to insert arbitrary code. To detect and analyze injected code (often called *shellcode*), some researchers have proposed network-level code emulators. A network-level code emulator emulates shellcode's behaviors and thus can detect shellcode accurately and help analysts understand the behaviors. Since it interprets shellcode, it is robust to encryption and obfuscation. We demonstrate that *memory-scanning attacks*, which use data on the victim process, can evade current network-level code emulators, and propose Yataglass+, an elaborated network-level code emulator, that enables us to analyze shellcode that incorporates memory-scanning attacks. Experimental results show that Yataglass+ successfully analyzes real shellcode into which we had manually incorporated memory-scanning attacks.

## 1. はじめに

現在，リモートコードインジェクション攻撃がいまだセキュリティ上の脅威であり続けている．リモートコードインジェクション攻撃では，攻撃者は様々なインターネットサーバの脆弱性を利用し攻撃コードを実行させることで被害を与える．リモートコードインジェクション攻撃のために利用できる脆弱性として，スタック溢れ脆弱性[1)]，ヒープ上書き脆弱性[2)]，フォーマット文字列脆弱性[3)] などがよく知られている．

一方，サーバ管理者はネットワーク侵入検知システム（NIDS）[4)–6)] を用いてリモートホストからの攻撃を検知している．NIDS でリモートコードインジェクション攻撃を検知する手法として，攻撃コード中のバイト列を調べたり[4),5)]，攻撃コードの制御フローを調べたりする手法[6)] が提案されている．しかし，攻撃者はこのような手法による攻撃コードの検知を回避するために，攻撃コードに暗号化や難読化を適用するようになってきた[7)–10)]．暗号化では攻撃コードが実行時に自身を書き換えながら実行するように攻撃コードの変換を行う．また，難読化では命令列の実行に影響しないよう余分なバイト列を命令と命令の間に挿入するなどの手法がある．

このような暗号化や難読化がなされた攻撃メッセージを解析するために，近年ではネットワーク・コードエミュレータが提案されている．ネットワーク・コードエミュレータでは，メッセージのバイト列を機械語命令列と見なして疑似実行を行うことでメッセージ中に含まれる攻撃コードを解析する．ここで，暗号化された攻撃コードを実行した場合は，疑似実行の過程で暗号化されたデータが復号される．また，難読化のために挿入されたバイト列は疑似実行に影響を与えない．したがって，攻撃者は暗号化や難読化によってネットワーク・コードエミュレータによる解析を回避することはできない．これまでに，Polychronakis ら[11),12)] と，Zhang ら[13)] はメッセージ中の攻撃コードを検知するためのネットワーク・コードエミュレータを提案している．また，攻撃コードの用いる Win32 API を抽出するネット

†1 慶應義塾大学理工学部情報工学科
Department of Information and Computer Science, Keio University
†2 科学技術振興機構 CREST
CREST, Japan Science and Technology Agency

⋆1 本論文は，国際会議 DIMVA 2009 に採録された以下の論文の内容を拡張したものである．“Yataglass: Network-level Code Emulation for Analyzing Memory-scanning Attacks”, Makoto Shimamura and Kenji Kono, July 2009．また本論文は情報処理学会論文誌 Vol.50, No.9 に掲載された以下の論文に基づいている．“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,” 嶋村　誠，河野健二，Sept. 2009.

ワーク・コードエミュレータとして，Spector[14] が提案されている．さらに，著者らは攻撃コードの用いる Linux システムコールと Win32 API を抽出するネットワーク・コードエミュレータである Yataglass を提案している[15]．

しかし，攻撃者はネットワーク・コードエミュレータによる解析を回避するために様々な手法を用いるようになっている[16]．たとえば，TAPiON エンコーダ[8] は攻撃コードに意味のない浮動小数点演算命令や `rdtsc` 命令を NOP の代わりに挿入することで，浮動小数点演算命令や `rdtsc` 命令を実装していないエミュレータによる解析を阻害する．Polychronakis ら[11] のエミュレータ，および Yataglass はこれらの命令をスキップすることで TAPiON でエンコードされた攻撃コードを正しく解析している．しかし，攻撃者は今後もエミュレータを回避するための技術を開発する可能性がある．もし，新しい回避技術が攻撃コードに使用されるようになると，攻撃コードの解析に既存のネットワーク・コードエミュレータが利用できなくなるため，解析者が人手で解析を行わなければならなくなる．このため，ネットワーク・コードエミュレータの持つ弱点について議論し，対策となる手法を未然に実現しておくことが重要である．

本論文では，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**[17] が既存のネットワーク・コードエミュレータの回避に利用できることを示す．そして，その回避手法の対策を行ったネットワーク・コードエミュレータである Yataglass+を提案する．メモリスキャン攻撃は攻撃を受ける被害プロセスのメモリにあるデータを攻撃コードの一部として利用する攻撃である．既存のネットワーク・コードエミュレータは被害プロセスのメモリ内容を用いることなく，攻撃コードのみを用いて解析を行う．そのため，メモリスキャン攻撃を用いる攻撃コードを正しく実行できない．メモリスキャン攻撃の例として，被害プロセスのメモリから Intel x86 の `ret` 命令である `0xC3` を探して，見つかったアドレスに対して `call` 命令で制御を移す攻撃が考えられる．この攻撃では，いったん攻撃コードから被害プロセスの命令列に制御が移るが，すぐに攻撃コードに制御が戻り，攻撃コードの続きを実行する．既存のネットワーク・コードエミュレータでは，被害プロセスのメモリにある命令列が分からないため，この攻撃コードの実行に失敗してしまう．Polychronakis ら[11] はこのような攻撃によってネットワーク・コードエミュレータが回避できる可能性を示唆しているものの，詳しい議論はしていない．本論文では，メモリスキャン攻撃が実際にネットワーク・コードエミュレータを回避できること，およびメモリスキャン攻撃が実際の攻撃コードに対して容易に適用できることを示す．

Yataglass+は著者らが提案した Yataglass[15] をメモリスキャン攻撃が解析できるように拡張したネットワーク・コードエミュレータである．Yataglass+では，メモリスキャン攻撃を解析するために，攻撃コードが利用する被害プロセス上のデータを攻撃コードの振舞いから抽出する．その後，Yataglass+は攻撃コードが利用するデータを用意し，その領域を被害プロセスのメモリ領域として攻撃コードに使用させる．このようにして，Yataglass+はメモリスキャン攻撃を解析し，攻撃コードがネットワーク・コードエミュレータを回避することを防止する．Yataglass+で用いるメモリスキャン攻撃への対策は既存のネットワーク・コードエミュレータに適用できるため，本研究によりネットワーク・コードエミュレータの有用性を高めることができる．Yataglass+を実現するには，攻撃コードが探す被害プロセス上のデータをどう抽出するかということが課題になる．このため，Yataglass+では攻撃コードの Symbolic Execution を行い，攻撃コードが被害プロセス中から探すデータの条件を推測する．詳しくは 4 章で説明する．

なお，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行う攻撃コードを解析する．もし，被害プロセスのメモリ内容を用いて解析を行うとなると，ネットワーク・コードエミュレータ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[18] 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る．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．したがって，Yataglass+は既存のネットワーク・コードエミュレータと同様に，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．

実際に Intel x86 の命令セットを疑似実行する Yataglass+のプロトタイプを実装し，有効性を示すため実験を行った．この実験では，最新のネットワーク・コードエミュレータである Spector[14] と Yataglass+との比較を行った．Spector はメモリスキャン攻撃に耐性を持たないため，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．一方，Yataglass+はそのような攻撃コードを解析できる．7 つの実際の攻撃コードに対しメモリスキャン攻撃を組み込んだところ，Spector は攻撃コードの解析に失敗し，システムコールを抽出することなく実行を停止した．一方，Yataglass+は攻撃コードを正しく解析でき，攻撃コードの呼び出すシステムコールを抽出できた．

本論文の構成は以下のとおりである．まず，2 章で Yataglass+の解析対象とする攻撃コードとネットワーク・コードエミュレータについて説明する．次に 3 章でメモリスキャン攻撃について説明する．4 章では Yataglass+の概要について示し，5 章で Yataglass+の疑似実行について説明する．6 章では Yataglass+にメモリスキャン攻撃を解析させた実験結果

について示す．7 章では Yataglass+の制限となる点について示す．8 章で関連研究をまとめる．最後に 9 章で本論文をまとめる．

## 2. 本研究の背景

### 2.1 攻撃コード

リモートコードインジェクション攻撃では，攻撃対象のサーバプログラムの脆弱性を利用し，攻撃メッセージ中の攻撃コードをサーバプログラムに挿入する．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．リモートコードインジェクション攻撃の中でも特にバッファ溢れ攻撃がよく知られている[1]．そのほかにも，フォーマット文字列攻撃[19]，ヒープの二重解放を利用した攻撃[20]，return-into-libc 攻撃[21] など，様々なリモートコードインジェクション攻撃の方法が存在している．

攻撃コードは攻撃の目的を達するために，攻撃対象の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．たとえば，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きわ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．

なお，近年の攻撃コード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多いため，攻撃の全体を解析するには，攻撃コードを解析するだけでは十分ではない可能性がある．しかし，このような場合でも攻撃コードを解析することでダウンロードするプログラムの入手元が分かる．そのため管理者はこの情報を用いてプログラムを入手したり，攻撃の対策を立てたりすることができる[14]．

### 2.2 ネットワーク・コードエミュレータ

#### 2.2.1 ネットワーク・コードエミュレータの動作

ネットワーク・コードエミュレータは，攻撃メッセージ中に含まれる攻撃コードを疑似的に実行し解析するシステムである．具体的には，ネットワーク・コードエミュレータは仮想的なレジスタとメモリを攻撃コードに従って操作することで攻撃コードを解析する．**表 1** に Intel x86 アーキテクチャの代表的なレジスタを示す．攻撃コードが攻撃メッセージ中のどの位置に存在するかは事前には分からないため，ネットワーク・コードエミュレータは攻撃メッセージ中の攻撃コードの位置を特定するために様々なアプローチを用いる．たとえば，Polychronakis らのエミュレータ[11] と Yataglass は攻撃メッセージのすべてのバイト位置から命令列の実行を試すことで攻撃コードを検索する．Spector[14] では攻撃コードの解析を行う前に NIDS を用いて攻撃コードの位置を特定する．

**表 1**　Intel x86 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
Table 1　Freqently used registers in Intel x86. Registers in lower half are not directly accessible in instruction operands.

| 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` の条件分岐に使用） |

ネットワーク・コードエミュレータには 2 つの利点がある．第 1 に，暗号化や難読化が施された攻撃コードを解析することができる．暗号化された攻撃コードはエミュレータで実行することにより復号化される．また，エミュレータによる解析結果は難読化の影響を受けない．第 2 に，実際にはサーバ上で成功しない攻撃コードを解析することができる．実際，Spector[14] はハニーポットで集めた攻撃コードの解析に用いられている．

ネットワーク・コードエミュレータによる疑似実行は 2 つの目的で利用されている．第 1 に，攻撃コードの振舞いを調査するために用いられる．たとえば，Spector[14] と Yataglass では，疑似実行の結果として，攻撃コードが実行した命令列と発行したシステムコールや Win32 API の呼び出しを出力する．管理者はこれらの結果を見ることで，攻撃コードの被害プロセス上での振舞いを理解できる．第 2 に，自己改変を行う暗号化された攻撃コードの検知に用いられる．たとえば，Polychronakis らのエミュレータ[11] と Zhang[13] らのエミュレータでは，1) 命令カウンタを取り出す *GetPC* と呼ばれる命令列を用いるかどうか，2) 自己改変を行うためにメッセージの中身を読み出すかどうか，という 2 点を用いて自己改変を行う攻撃コードを探す．

#### 2.2.2 ネットワーク・コードエミュレータの回避可能性

攻撃者はネットワーク・コードエミュレータを回避するために様々な手法を開発している．もし，新しい回避技術が攻撃コードに使用されると，従来のネットワーク・コードエミュレータによる攻撃コードの解析は回避されてしまう．したがって，暗号化や難読化がなされた攻撃コードやハニーポットで集めた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．

攻撃者はネットワーク・コードエミュレータによる解析を大きく分けて 2 つの手法で回避できる．第 1 に，攻撃コードはエミュレータが実装していない機能を利用することでエミュレータによる解析を回避できる．たとえば，書式が既知のファイル（サーバの設定ファイルなど）を読み出し，書式が予想と異なる場合，それ以降の動作を停止する攻撃コードが考えられる．この場合，エミュレータは書式チェック以降の攻撃コードの振舞いを抽出できない．この回避手法はネットワーク・コードエミュレータの回避手法としてすでに用いられている[8),16)]．この回避手法を防止するにはマルウェア解析を行うためのエミュレータで用いられている手法[22)–24)] を利用し，書式チェックの結果を利用した条件分岐命令において，分岐の両方を実行することで実行を続ければよい．この手法は Yataglass にすでに実装済みである[15)] ため，本論文ではこれ以上の議論は行わない．

第 2 に，攻撃コードは被害プロセス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エミュレータによる解析を回避できる．この場合，エミュレータは被害プロセスのメモリ内容を持たないため，実行に失敗する．メモリスキャン攻撃[17)] は被害プロセス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．本論文ではこの攻撃への対策を行ったエミュレータを提案する．

## 3. メモリスキャン攻撃

メモリスキャン攻撃では，被害プロセス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．攻撃者は以下の 2 つの方法で被害プロセスの持つデータを使用できる．

- **既知のアドレスにあるデータの利用**　攻撃コードは，被害プロセス中のデータのアドレスを指定することで，被害プロセス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なければならない．Polychronakis ら[11)] は，この攻撃は被害プロセス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．また，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，まず初めに被害プロセス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，7 章で議論する．
- **被害プロセス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，被害プロセス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[17)] 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ことを示した．しかし，さらにこの攻撃はネットワーク・コードエミュレータを回避するために使われる可能性がある．この攻撃は被害プロセス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成できる．もしそのようなツールが作成された場合，既存のネットワーク・コードエミュレータは容易に回避されるようになる．したがって，本論文ではこのタイプの攻撃に着目する．

最も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被害プロセス中のデータとして利用する．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`）から容易に得られるためである．しかし，スタック領域は実行禁止になっていることが多いため，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．また，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．

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 1 に，コード領域のデータは，つねに実行可能であり，攻撃コードが実行可能な命令列を探すことができる．第 2 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．

攻撃者は，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．たとえば，`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[25)] 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．攻撃者は被害プロセスのメモリ配置をすべ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，被害プロセス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 1** に示す．この攻撃は，`ret` 命令（`0xC3`）を被害プロセスのメモリから探し，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 つの手続きからなる．まず，GetPC と呼ばれる手続きにより，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．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．たとえば，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，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被害プロセス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被害プロセス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．

メモリスキャン攻撃は既存のネットワーク・コードエミュレータによる疑似実行を回避することができる．実際，最新のネットワーク・コードエミュレータである Spector[14)]，Polychronakis ら[11),12)] のエミュレータ，Zhang ら[13)] のエミュレータでは，エミュレーション時に被害プロセスのメモリ領域は参照していない．このため，攻撃コードが被害プロセス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，エミュレータ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，攻撃コードは被害プロセス中のデータを用いて命令を実行するため，攻撃コードが正しく実行されたときの状態とエミュレータの状態が異なってしまう．したがって，エミュレータはメモリスキャン攻撃より後の命令列を正しく実行できない．

既存のネットワーク・コードエミュレータによる攻撃コードの解析はメモリスキャン攻撃を攻撃コード中の適切な場所に挿入することで回避されてしまう．たとえば，図 1 では，GetPC と復号ループの間に回避コードを挿入している．この場合，GetPC を用いて攻撃コードを検出するエミュレータ[11),13)] は攻撃コードを検知できない．エミュレータが GetPC から攻撃コードを実行しはじめた場合は，メモリスキャン攻撃によって実行が中断される．一方，復号ループから実行をはじめた場合は，GetPC を行っていないため，攻撃コード中の暗号化されたデータを読み出せない．

また，攻撃者がメモリスキャン攻撃をシステムコールの呼び出しの前に挿入すると，攻撃コードのシステムコール呼び出しを抽出するエミュレータ[14),15)] は攻撃コードを解析することができなくなる．これは，エミュレータがメモリスキャン攻撃以降のコードを実行しても，システムコールに与える引数を解析できないためである．

さらにメモリスキャン攻撃を用いて被害プロセス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たりすることができる．たとえば，攻撃コードは被害プロセス中の `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，およびその引数を解析できない．

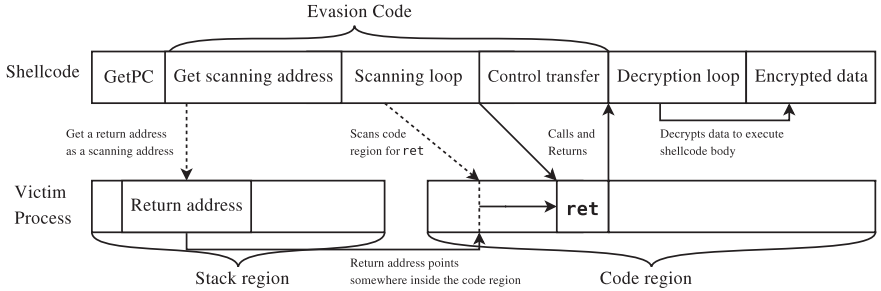


**図 1**　メモリスキャン攻撃の例
Fig. 1　Example of memory-scanning attack.

## 4. Yataglass+

本論文では，メモリスキャン攻撃に対策を行ったネットワーク・コードエミュレータである Yataglass+を提案する．Yataglass+は，著者らの提案したネットワーク・コードエミュレータである Yataglass[15)] をメモリスキャン攻撃によって回避されないよう拡張したものである．**図 2** に Yataglass+の全体像を示す．Yataglass+は Spector[14)] と同様に攻撃コードを疑似的に実行し，攻撃コードの解析結果として，疑似実行した命令列と攻撃コードの

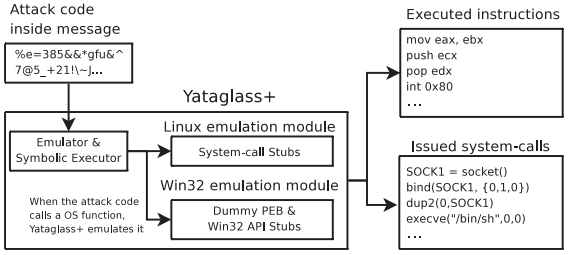


**図 2**　Yataglass+の概要
Fig. 2　Overview of Yataglass+.

用いるシステムコール列を出力する．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エミュレーションについては 4.1 節，Win32 API のエミュレーションについては 4.2 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舞いを解析するには，既存のマルウェア解析技術[23),26)] を用いればよい．

Yataglass は Spector と 2 つの点で異なる．第 1 に，Yataglass は Windows と Linux の両方の攻撃コードに対して振舞いを解析できる．一方，Spector は Win32 API しか解析できない．第 2 に，Yataglass は Spector と比較して実行速度の面で優れている．これは Spector が XOR などのビット演算の場合に，各ビットに対して演算を行い，各ビットの演算履歴を追跡する一方，Yataglass はそれをしないためである．このため，Yataglass は Intel Core2 Duo 1.86 GHz 上で毎秒 26,773 命令実行できる．一方，Spector ではプロセッサ種別は明らかにされていないものの，Intel 製の 2 GHz の CPU 上で毎秒 842 命令しか実行できないと述べられている[14)]．なお，Yataglass および Yataglass+ではビットごとの演算履歴を追跡しないため，ビット演算の結果を簡潔な形で表せないことがある．この制限については 7 章で詳しく述べる．

メモリスキャン攻撃に対応するため，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 [14),22)] を用いて被害プロセス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被害プロセス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．たとえば，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実行を続ける．これについては，5.2 節で詳しく述べる．なお，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．しかし，メモリスキャン攻撃では攻撃コードが被害プロセス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．たとえば，`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` 命令を被害プロセス中から探すものの，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 7 章で詳しく述べる．

Yataglass+は初期化時にレジスタとスタックとして用いるメモリ領域を初期化し，攻撃コードをメモリに読み込む．その後，Yataglass+は攻撃コードの実行を，`exit()` などの被害プロセスを終了させるシステムコールや，`execve()` などの他のプログラムを起動し，制御を移すシステムコールが発行されるまで行う．Yataglass+が攻撃コードを実行している途中で `int 0x80` 命令や `sysenter` 命令のようなシステムコールの呼び出しを発見した場合には，システムコールの名前と引数を記録する．以下，本章では Yataglass+に実装されている Yataglass の基本的な疑似実行の仕組みについて述べ，次章で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．

### 4.1　システムコールの疑似実行

Yataglass+はシステムコールの実行を検知した場合，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+が攻撃コードの解析を続けられるようにするために呼び出す処理である．たとえば，`open()` に対するスタブでは，ディスクリプタ番号を生成し攻撃コードに返す処理を行う．これにより，攻撃コードは `open()` が成功したと考え，実行を続ける．しかし，攻撃コードはこれらのシステムコールの返り値を調べることでエミュレータ上で実行されていることを検知し，実行を停止する場合がある．たとえば，実在しないファイルの `open()` を試みたにもかかわらず，`open()` が成功すれば，攻撃コードがエミュレータ上で実行されていることが分かる．このため，Yataglass+は，他のエミュレータで用いられている対策[22)–24)]（2.2.2 項を参照のこと）を用いることで，この

ような攻撃に対処している．具体的には，Yataglass+はシステムコールが呼び出されたときに，システムコールの返り値として用いられる eax レジスタのデータに偽のデータであるというフラグを設定し，このデータを用いた演算を追跡する．そして，偽のデータに基づく条件分岐を検出した場合，条件分岐が成立した場合とそうでない場合の両方を解析する．これにより，システムコールの成功失敗にかかわらず，攻撃コードの解析を続けることができる．

また，`read()` や `recv()` のように，バッファに外部データを読み込むシステムコールがある．現在の Yataglass+では，バッファの内容を特に書き換えることなく実行を続ける．これは，書き込むべきデータが攻撃者に由来するものである場合，そのデータがまだ入手できていないかもしれないためである．このようにすることで，Yataglass+はデータが入手できていない場合でも攻撃コードの解析をできるだけ行うようにしている．ただし，攻撃コードがこれらのシステムコールを実行した後に，バッファ内容を調べることで動作を変更する可能性がある．このため，Yataglass+はバッファの中身に偽のデータであるというフラグを設定することで，システムコールの返り値と同様に，このバッファの中身に基づく条件分岐を発見したときに，この条件分岐が成立した場合とそうでない場合の実行を行う．このようにすることで，バッファの内容が実際攻撃コードを実行するときのバッファの内容と異なっていたとしても，攻撃コードの解析を行うことができる．なお，攻撃コードが取得したバッファの中身を復号化キーや命令列として用いる場合がある．この場合，現在の Yataglass+は攻撃コードの解析を行えない．しかし，この場合には，これらのシステムコールのスタブを拡張し，外部のデータを用いてバッファの中身を書き換え，解析を続ければよい．

Yataglass+はスタブを用いることで，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いでいる．たとえば，`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は，実際に Yataglass+による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続けなければならない．これは攻撃コードにおいて親プロセスと子プロセスで振舞いが異な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で 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ている．また，`send()` のような外部と通信を行うシステムコールは，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．したがって，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 eax レジスタのデータに偽のデータであるというフラグを設定する．

現在の Yataglass+では，攻撃コードが用いる可能性の低いシステムコールや，返り値以外に攻撃コードに影響を及ぼす可能性が少ないと考えられるシステムコールについては汎用のスタブを用いている．この汎用のスタブでは eax レジスタのデータに偽のデータであるというフラグを設定する処理を行う．たとえば，`chdir()` は Yataglass+の実行結果に影響を及ぼす可能性が少ないため，汎用のスタブで処理できる．`chdir()` はプロセスの作業ディレクトリを変更し，`open()` の結果に影響を与えるシステムコールである．しかし，Yataglass+では，攻撃コードが `open()` を呼び出した場合に，実際のファイルの有無にかかわらず Yataglass+が生成した偽のディスクリプタ番号を返すようになっているため，作業ディレクトリの影響を受けない．

### 4.2　Win32 API への対応

Windows に対する攻撃コードはシステムコールに加えて，Win32 API を用いて計算機資源へのアクセスを行うことで，サーバに被害を及ぼす．しかし，Win32 API を用いる攻撃コードは単純にメッセージ中の命令列を解析しスタックやレジスタの内容を調べるだけでは正しく解析できない．これは，このような攻撃コードが被害プロセスのメモリ上に存在する Process Environment Block（PEB）と，動的リンクライブラリ（DLL）中の API を利用するためである．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 のリストを用いて，DLL に存在する API を用いることができる．たとえば，`ws2_32.dll` はネットワークソケットを扱う DLL であり，被害プロセスにロードされていることが多い．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 上のプロセスはつねに `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()` によって被害プロセスに新しい 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，被害プロセスにインストールされた DLL の API を利用する．したがって，このような攻撃コードの解析を行うには，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スタブを用意したりして，疑似的に API を実行する必要がある．

したがって，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，攻撃コードが PEB のあるアドレスである `0x7FFDF000` 番地，もしくは FS セグメントからデータを読み出したときに，ダミーの PEB を見せるようにする．具体的には，Yataglass+が攻

撃コードに見せているメモリ領域に kernel32.dll，user32.dll，ws2_32.dll などのよく利用される DLL を展開し，それらの DLL のアドレスを保持するリストを作り，PEB として攻撃コードに見せる．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 をメモリに展開することにより DLL の持つ API のアドレスが確定するので，これらの API の呼び出しを検出できる．これにより，Yataglass+は LoadLibrary() や GetProcAddress() といった API の呼び出しを検出し，これらの API を疑似的に実行することができるようになる．このようにダミーの PEB を攻撃コードに使用させることで，Win32 API を呼び出す攻撃コードの多くを実行できるようになる．なお，PEB の内容を検査しダミーかどうか判断する攻撃コードも考えられるため，その対策として PEB で用いるデータはすべて偽のデータであるというフラグを与えている．4.1 節で述べたとおり，Yataglass+は攻撃コードが偽のデータに由来する条件分岐を行った場合，その条件分岐が成立した場合とそうでない場合を両方実行することで解析を続けることができる．しかし，Yataglass+では PEB のデータを正確に再現しているわけではない．このため，攻撃者があらかじめ被害プロセスの PEB の保持する値を調べておいて，この値を攻撃コードの復号化キーとして用いる場合，Yataglass+では攻撃コードの解析ができない．

Yataglass+において，LoadLibrary() や GetProcAddress() 以外のあまり使われない Win32 API では，共通のスタブとして，スタックから引数を除去し，eax レジスタのデータに偽のデータであるというフラグを設定するこれは，攻撃コードが LoadLibrary() や GetProcAddress() を用いて，任意の DLL の API を呼び出すことができる一方で，Yataglass+では攻撃コードの利用するすべての Win32 API についてスタブを用意することが難しいためである．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，以降の解析が行えなくなる．このため，Yataglass+では DLL の中の実行コードを API のアドレスから静的に解析していき，初めに見つけた retn 命令の引数を用いることでスタックから引数を除去する．これにより，スタックポインタを正しく元に戻して解析を続けることができる．

## 5. Yataglass+の疑似実行エンジン

### 5.1 Symbolic Execution

Yataglass+は，拡張前の Yataglass と異なり，攻撃コードの Symbolic Execution [14),22)] を行う．Symbolic Execution とは，エミュレータ内のすべ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．たとえば，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 1 つのシンボルになる．ここで，このシンボルが持つ値は実際に *X* と *Y* の値が分かった場合に初めて決定される．拡張前の Yataglass では Symbolic Execution を実装していないため，メモリスキャン攻撃によって回避されてしまうようになっていた．

Value には，*Number*，*Symbol*，*Expression*，*Unknown* の 4 種類がある．Number は数値を表す種類の Value であり，具体的な確定している値を示す．たとえば 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 つからなる 3 つ組で，たとえば 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．たとえば，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 の初期状態は分からないが，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．たとえば，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．たとえば，*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うえで重要となる．5.2 節では，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・ループによる被害プロセス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，3 章で説明したとおり，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したりする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，被害プロセス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．たとえば，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* を参照した場合，被害プロセス中のメモリ領域をアクセスしたと見なし，取得したデータに，

*CODE*$_n$ という Symbol を与える．なお，ここで $n$ は被害プロセス中のメモリ領域の $n$ バイト目から取得したデータという意味である．

Yataglass+は，他の Symbolic Execution システム[14),22)]と同様に，すべ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．たとえば，TAPiON[8)] 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．したがって，Yataglass+は，2 つの Value である *X* と *Y* について以下のように縮約する．

- *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．たとえば，$X = 2$，$Y = 1$ であることが分かっているとき，(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．たとえば，(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　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被害プロセス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[14)] 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．しかし，エミュレータ上において，メモリスキャン攻撃は被害プロセス中のメモリ状態が分からないため非決定的な動作をする．**図 3** 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* で示される被害プロセス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

```
# 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 3**　スキャニング・ループの例
Fig. 3　Example of scanning loop.

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() によって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 1 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 3 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．

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 3 の例では，4 行目の cmp 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 (*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 3 の例では，Yataglass+は，5 行目で *CODE*$_1$（[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

```
# ADDR is an address of code region
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 4**　2 つの条件を用いるスキャニング・ループの例
Fig. 4　Example of scanning loop that uses two constraints.

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，攻撃コードに被害プロセス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，$CODE_1$（`[edi]`）の値は `0xC3` になる．したがって，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 4** 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，$CODE_1$（`[edi]`）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，$CODE_1$（`[edi]`）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．たとえば，`repne scasb` 命令は，`edi` が指すメモリ領域から，`eax` の下位 1 バイト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，`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．

### 5.3 実　　装

Yataglass+のプロトタイプを Linux 上に実装した．x86 命令のデコードには，libdasm [27] を用いた．エミュレータは 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．また，fstenv，fnstenv，fsave，fnsave を除く FPU，SIMD，特権命令は実行しない．これらの命令は現状の攻撃コードではほとんど用いられていない．Yataglass+では実装を簡略化するため，これらの命令は疑似実行せずにスキップしている．

また，4.2 節で述べたとおり，Windows に対する攻撃コードを解析するために，偽の Process Environment Block（PEB）を用意した．具体的には，Yataglass+の初期化時に，攻撃コードが使用する DLL である `kernel32.dll`，`user32.dll`，`ws2_32.dll` を展開し，攻撃コードがこのデータを使い API のアドレスを探す場合を扱えるようにした．さらに，`LoadLibrary()` や `GetProcAddress()` など，代表的な Win32 API のスタブを実装した．これにより，`LoadLibrary()` や `GetProcAddress()` を用いる攻撃コードを解析できる．

## 6. 実　　験

### 6.1 メモリスキャン攻撃を利用する攻撃コードの作成

本章では，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．また，Spector [14] を独自に実装したものを用いて Yataglass+との比較を行った．Spector がシステムコールを抽出する能力は Yataglass+と同じだが，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被害プロセス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 以外のネットワーク・コードエミュレータとして，Polychronakis ら [11),12)] や Zhang ら [13)] のエミュレータがある．しかし，これらのエミュレータはすべて攻撃コードの解析をメッセージのみを用いて行っており，被害プロセス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 と同じになると考えられる．したがって，本論文では Yataglass+とこれらのネットワーク・コード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

**表 2**　実験で用いた攻撃コード
Table 2　Shellcodes used in the experiments.

| 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 |

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[28]，wu-ftpd[29]，rsync[30]，wu-imap[31]，Apache Web Server[20]，samba[32]，cyrus-pop3d[33] である．**表 2** 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[34] と Milw0rm[35] 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，被害プロセス中の命令列から `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，被害プロセス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` レジスタを破壊する．このため，被害プロセス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` 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．また，wu-imap に対する脆弱性では，wu-imap が `ToUpper()` 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，攻撃コード中の小文字を動的に生成するコードを作成した．

**表 3**　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
Table 3　Summary of emulation results. √ means that the emulator successfully analyzed the code, and − means that it failed to analyze the code.

| 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） | √ | − |

| ファイル名 | Yataglass+ | Spector-X |
|---|---|---|
| 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） | √ | − |

### 6.2　メモリスキャン攻撃を利用する攻撃コードの解析

表 2 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 3** 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．すべ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** 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分かる（命令番号 `004d` から `0067`）．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．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．Yataglass+はこの比較ループを `eax` に制約条件をつけて抜ける．その後，攻撃コードは，被害プロセス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得る（攻撃コードの `00d5` 番地）．また，攻撃コードはスキャニング・ループにより，被害プロセス中の命令列から `pop ebp` と `ret` の並びを探す．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命令列（*`CODE_1`*）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，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 `execve()`

```
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
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 c3
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**　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．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

Fig. 5　The emulation result of shellcode generated by rsync-expl.c, incorporated with memory-scanning attack, with Yataglass+. Logs of first 64 (0x40) instructions and outputs from forked instances are omitted.

```
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 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 Spector-X による解析結果．初めの 64 個（0x40）の命令は省略した

Fig. 6　Emulation result of shellcode generated by rsync-expl.c, incorporated with memory-scanning attack, with Spector-X. First 64 (0x40) instructions are omitted.

システムコールを用いて，/bin/sh を実行する．

**図 6** 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 は攻撃コードが被害プロセス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）．したがって，Spector-X はこの攻撃コードを正しく実行できていない．

## 7. 議　　論

Yataglass+では，以上に示したように，メモリスキャン攻撃を用いてネットワーク・コードエミュレータを回避する攻撃コードを解析できた．本章では，Yataglass+の制限となる点と，それを利用して Yataglass+を回避する可能性のある攻撃について述べる．

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．したがって，攻撃者は他の方法でコード領域のアドレスを得ることで，Yataglass+を回避する可能性がある．たとえば，攻撃コード中で，/proc/XXX/maps（XXX は被害プロセスのプロセス 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．この場合は，Yataglass+が偽の GOT や PLT を持たせればよい．

また，今後，Yataglass+はより優れたスキャニング・ループに対処しなければならない．たとえば，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 命令のどれでもよいように作成し，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 がとる値の数がある閾値を切ったときに，それぞれの値を仮定し実行すればよい．10 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と考えられる．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．

4 章で述べたとおり，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．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．たとえば，Linn ら[17] 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．この場合，Yataglass+は 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．

また，Yataglass+は攻撃者による 2 段階の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，被害プロセス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 2 の攻撃コードはメモリをスキャンせず被害プロセス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 1 の攻撃コードを実行した後，その状態を保存したまま，第 2 の攻撃コードを実行できるようにする必要がある．

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．たとえば，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，実際の攻撃コードはその大きさ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．

最後に，Yataglass+は攻撃者が被害プロセス中の制御変数を書き換えて実行を変えてしまうという攻撃には対処できない．これは，Yataglass+が被害プロセス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．3 章でも述べたように，攻撃コードは被害プロセス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 [36] などの技術が広く用いられているためである．したがって，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．

## 8. 関 連 研 究

ネットワーク・コードエミュレータについてはすでに 2 章で述べているため，本章ではその他の関連研究についてまとめる．

### 8.1 静 的 解 析

SigFree [37] は，メッセージの命令列としての長さを調べることで攻撃コードを検知する．SigFree は，メッセージのすべてのバイト位置から逆アセンブルを行い，実行可能なバイト列の長さが閾値を超えた場合，そのメッセージを攻撃メッセージであると見なす．Andersson ら [38] は攻撃コード中の `eax` レジスタへの代入と `int 0x80` 命令を検出することで，攻撃コードを検知する手法を提案している．しかし，これらの手法は静的に攻撃メッセージを解析するため，暗号化や難読化に弱い．Yataglass+ではこれらのシステムとは異なり，疑似実行により攻撃コードの振舞いを解析する．このため，Yataglass+は暗号化や難読化に強い．

また，Toth ら [39] は，abstract payload execution によって，攻撃メッセージ中の実行可能な命令列の数を調べ，NOP-sled と呼ばれる攻撃コード本体の直前に置かれ，攻撃コードの実行に影響を与えない命令列を検知する手法を提案している．しかし，Andersson らは，Windows のソフトウェアはバイナリで配布されているため，Windows に対する攻撃コードには NOP-sled は必要ないと述べている [40]．

Kruegel ら [41] のワーム検出手法では，メッセージを静的に解析して，実行可能命令列の検出を行う．この手法では，メッセージを命令列だと仮定して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成することで，命令列であるかどうかを判定する．そして，同様の制御フローグラフが多数のネットワーク接続から得られた場合，ワームの感染行動が発生していると見なす．対して，Yataglass+はワームによって実行される攻撃コードの解析を行う点で異なる．これらの手法は管理者が相互補完的に用いることができる．

STILL [42] は，静的解析ではあるものの，暗号化や難読化に強い解析手法である．STILL はまずメッセージを逆アセンブルし，制御フローグラフを作成する．次に，作成したグラフを用いて静的な Taint Analysis と Initialization Analysis を行い，暗号化された攻撃コードの復号エンジンや，難読化された攻撃コードが用いる間接ジャンプを検出する．STILL も Yataglass+も暗号化，復号化に強い．しかし，Yataglass+は攻撃コードの振舞いを抽出することが目的であるため，攻撃コードの検知を目的とする STILL とは異なる．

### 8.2 ホスト侵入検知システム

ホスト侵入検知システムでは，攻撃コードがサーバで実際に発生した振舞いを検出することができる．たとえば，Linn ら [17] 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とができる．これは，プログラム中のすべ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．これにより，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 [43] は仮想マシンを用いた HIDS 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，ハードウェア割込みなどの非決定性のあるイベントをすべてロギングしておく．これにより，サーバが攻撃されたことを検出した後で，攻撃の実行過程をすべて再生することができる．これらの HIDS はすでに攻撃されたサーバを調べるものであ

る．対して，Yataglass+は攻撃コードを解析する．したがって，Yataglass+は HIDS とは異なり，失敗した攻撃についても解析することができる．このため，Yataglass+はハニーポットで集めた攻撃コードについても解析することができる．

Andersson ら[40] は攻撃コードをサンドボックス上のプロセスに注入し実行する手法を提案している．このシステムでは，攻撃コードによって実行されたシステムコール列を取得できる．しかし，攻撃者はこのシステムを，システムコールの結果を調べることで回避できる．Yataglass+はネットワーク・コードエミュレータであるため，4.1 節で述べたように，システムコールの結果を調べるような条件分岐を検出し，その分岐を両方たどることで，チェックを成功させたかのように攻撃コードを実行することができる[22)–24)]．対して，このシステムは CPU 上で直接攻撃コードの命令列を実行してしまうため，そのような条件分岐を検出することが難しい．

## 9. 終わりに

リモートコードインジェクション攻撃では，サーバの脆弱性を利用し，攻撃コードと呼ばれる機械語命令列をサーバに挿入し実行させる．この攻撃コードを検知・解析するシステムとして，攻撃コードの疑似実行を行うことで攻撃コードを検知・解析するネットワーク・コードエミュレータが提案されている．本論文では，既存のネットワーク・コードエミュレータは被害プロセスのメモリ上のデータを命令列として利用するメモリスキャン攻撃により回避できることを示し，また，メモリスキャン攻撃を利用した攻撃コードの振舞いを解析するネットワーク・コードエミュレータである Yataglass+を提案した．実際に Yataglass+のプロトタイプを作成し，実際の攻撃コードにメモリスキャン攻撃を適用し実験を行った結果，Yataglass+は正しくメモリスキャン攻撃を適用した攻撃コードを解析できた．今後の課題としては，より高度なメモリスキャン攻撃への対策を考える必要がある．また Yataglass+の応用として，Yataglass+の解析結果を用いて攻撃の被害からの復旧を支援するシステムや，Yataglass+とシステムコールベースの侵入検知システムを連携させることが考えられる．

## 参考文献


1) AlephOne: Smashing stack for fun and profit, Phrack (1996).
2) Sotirov, A.: Apache OpenSSL heap overflow exploit (2002). http://www.phreedom.org/research/exploits/apache-openssl/
3) Li, W. and cher Chiueh, T.: Automated Format String Attack Prevention for Win32/X86 Binaries, *Proc. 23rd Annual Computer Security Applications Conference* (*ACSAC '07*), pp.398–409 (2007).
4) Roesch, M.: Snort: Lightweight Intrusion Detection for Networks, *Proc. 13th USENIX Conference on Systems Administration* (*LISA '99*), pp.229–238 (1999).
5) Paxson, V.: Bro: A system for detecting network intruders in real-time, *Computer Networks*, Vol.31, No.23-24, pp.2435–2463 (1999).
6) Chinchani, R. and Berg, E.V.D.: A Fast Static Analysis Approach to Detect Exploit Code in Network Flows, *Proc. 8th International Symposium on Recent Advances in Intrusion Detection* (*RAID '05*), pp.284–308 (2005).
7) The Metasploit Project: Metasploit. http://www.metasploit.com/
8) Bania, P.: TAPiON (2005). http://pb.specialised.info/all/tapion/
9) K2: ADMMutate (2007). http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz
10) Sedalo, M.: JempiSCode (2006). http://goodfellas.shellcode.com.ar/proyectos.html
11) Polychronakis, M., Anagnostakis, K.G. and Markatos, E.P.: Network-Level Polymorphic Shellcode Detection Using Emulation, *Proc. 3rd Conference on Detection of Intrusions, Malware, and Vulnerability Assessment* (*DIMVA '06*), pp.54–73 (2006).
12) Polychronakis, M., Anagnostakis, K.G. and Markatos, E.P.: Emulation-Based Detection of Non-self-contained Polymorphic Shellcode, *Proc. 10th International Symposium on Recent Advances in Intrusion Detection* (*RAID '07*), pp.87–106 (2007).
13) Zhang, Q., Reeves, D.S., Ning, P. and Purushothaman-lyer, S.: Analyzing Network Traffic To Detect Self-Decrypting Exploit Code, *Proc. 2nd ASIAN ACM Symbopsium on Information, Computer and Communications Security* (*ASIACCS '07*), pp.4–12 (2007).
14) Borders, K., Prakash, A. and Zielinski, M.: Spector: Automatically Analyzing Shell Code, *Proc. 23rd Annual Computer Security Applications Conference* (*ACSAC '07*), pp.501–514 (2007).
15) 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 (2009).
16) Bania, P.: Evading network-level emulation (2009). http://piotrbania.com/all/articles/pbania-evading-nemu2009.pdf
17) Linn, C.M., Rajagopalan, M., Baker, S., Collberg, C., Debray, S.K. and Hartman, J.: Protecting Against Unexpected System Calls, *Proc. 13th Usenix Security Symposium*, pp.239–254 (2005).
18) Provos, N.: A Virtual Honeypot Framework, *Proc. 13th Usenix Security Symposium*, pp.1–14 (2004).
19) MITRE: Wu-Ftpd Remote Format String Stack Overwrite Vulnerability (2000). http://www.securityfocus.com/bid/1387

20) MITRE: OpenSSL SSLv2 Malformed Client Key Remote Buffer Overflow Vulnerability (2002). http://www.securityfocus.com/bid/5363
21) Nergal: The Advanced Return-into-lib (c) Exploits, *www. Phrack. org.*, Vol.58, No.4 (2001).
22) Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L. and Engler, D.R.: EXE: Automatically Generating Inputs of Death, *Proc. 13th ACM Conference on Computer and Communications Security* (*CCS '06*), pp.322–335 (2006).
23) Moser, A., Kruegel, C. and Kirda, E.: Exploring Multiple Execution Paths for Malware Analysis, *Proc. 2007 IEEE Symposium on Security and Privacy* (*S&P '07*), pp.231–245 (2007).
24) Brumley, D., Hartwig, C., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Song, D. and Yin, H.: BitScope: Automatically Dissecting Malicious Binaries, Technical Report CMU-CS-07-133, Carnegie Mellon University (2007).
25) SystemV Application Binary Interface Intel 386 Architecture Processor Supplement. http://www.caldera.com/developers/devspecs/abi386-4.pdf
26) Lanzi, A., Sharif, M. and Lee, W.: K-Tracer: A System for Extracting Kernel Malware Behavior, *Proc. 16th Annual Network and Distributed System Security Symposium* (*NDSS '09*) (2009).
27) jt: Libdasm (2006). http://www.klake.org/~jt/misc/libdasm-1.5.tar.gz
28) MITRE: ISC Bind 8 Transaction Signatures Buffer Overflow Vulnerability (2001). http://www.securityfocus.com/bid/2302
29) MITRE: Wu-Ftpd File Globbing Heap Corruption Vulnerability (2001). http://securityfocus.com/bid/3581
30) MITRE: Rsync Signed Array Index Remote Code Execution Vulnerability (2002). http://www.securityfocus.com/bid/3958
31) MITRE: Wu-imapd Partial Mailbox Attribute Remote Buffer Overflow Vulnerability (2002). http://securityfocus.com/bid/4713
32) MITRE: Samba 'call_trans2open' Remote Buffer Overflow Vulnerability (2003). http://securityfocus.com/bid/7294
33) MITRE: Cyrus IMAPD POP3D Remote Buffer Overflow Vulnerability (2006). http://www.securityfocus.com/bid/18506
34) SecurityFocus. http://securityfocus.com/
35) Milw0rm. http://www.milw0rm.com/
36) PaX Team: PaX address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt
37) Wang, X., Pan, C.-C., Liu, P. and Zhu, S.: SigFree: A Signature-free Buffer Overflow Attack Blocker, *Proc. 15th Usenix Security Symposium*, pp.225–240 (2006).
38) Andersson, S., Clark, A. and Mohay, G.M.: Network-Based Buffer Overflow Detection by Exploit Code Analysis, *Proc. AusCERT Asia Pacific Information Technology Security Conference*, pp.39–53 (2004).
39) Toth, T. and Kruegel, C.: Accurate Buffer Overflow Detection via Abstract Payload Execution, *Proc. 5th International Symposium on Recent Advances in Intrusion Detection* (*RAID '02*), pp.274–291 (2002).
40) Andersson, S., Clark, A., Mohay, G.M., Schatz, B. and Zimmermann, J.: A Framework for Detecting Network-based Code Injection Attacks Targeting Windows and UNIX, *Proc. 21st Annual Computer Security Applications Conference* (*ACSAC '05*), pp.49–58 (2005).
41) Kruegel, C., Kirda, E., Mutz, D., Robertson, W. and Vigna, G.: Polymorphic Worm Detection Using Structural Information of Executables, *Proc. 8th International Symposium on Recent Advances in Intrusion Detection* (*RAID '05*), pp.207–226 (2005).
42) Wang, X., Jhi, Y.-C., Zhu, S. and Liu, P.: STILL: Exploit Code Detection via Static Taint and Initialization, *Proc. 24th Annual Computer Security Applications Conference* (*ACSAC '08*), pp.289–298 (2008).
43) Dunlap, G.W., King, S.T., Cinar, S., Basrai, M. and Chen, P.M.: ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay, *Proc. 5th Symbosium on Operating Systems Design and Implementation* (*OSDI '02*), pp.211–224 (2002).

(平成 21 年 5 月 11 日受付)
(平成 21 年 9 月 9 日採録)

**嶋村　誠**（学生会員）

1982 年生．2005 年電気通信大学情報工学科卒業．2007 年慶應義塾大学大学院理工学研究科開放環境科学専攻修士課程修了．現在，同専攻博士課程在学中．オペレーティングシステム，システムソフトウェア，インターネットセキュリティに興味を持つ．IEEE，ACM 各学生会員．

**河野　健二**（正会員）

1993 年東京大学理学部情報科学科卒業．1997 年東京大学大学院理学系研究科情報科学専攻博士課程中退，同専攻助手に就任．現在，慶應義塾大学理工学部情報工学科准教授．博士（理学）．平成 11 年度情報処理学会論文賞受賞．平成 12 年度山下記念研究賞受賞．オペレーティングシステム，システムソフトウェア，インターネットセキュリティに興味を持つ．IEEE/CS，ACM，USENIX 各会員．