自己隠蔽挙動に基づいたIoTボットの振る舞い検知

田島 裕也[†1,a)]　小出 洋[†2,b)]

概要：本研究では IoT ボットの検知を効率的に行い，かつ誤検知が少なくなる方式を提案する．マルウェア検知で一般的なパターンマッチングは未知ボットや暗号化ボットは検知することが出来ない．これらのボットを検知することができる手法としてビヘイビア法が存在するが，誤検知が多いことや，ボット自体が他のマルウェアに比べて目立った挙動を起こさず，且つ自己の存在を隠蔽しようとするため検知しづらいといった問題がある．本研究ではこれらのボットに対するビヘイビア法の課題を解決するため，自己の存在を隠蔽する挙動を逆用した検知手法を提案し，検査プログラムを実装し，有効性を実験により確認した．

キーワード：コンピュータウイルス，侵入検出・検知，セキュア設計・実装

IoT bots detection based on stealth-behavior

Abstract: In this research, we propose a method to efficiently detect IoT bots and reduce false detections. General pattern matching cannot detect unknown bots or encrypted bots in malware detection. Behavioral methods exist that can detect these bots. However, there are many false positives, and the bots themselves do not behave more clearly than other malware, and try to conceal their existence. Therefore, there is a problem that it is difficult to detect. In this study, in order to solve the problems of the behavior method for these bots, we proposed a detection method that reverses the behavior of concealing the existence of self, implemented an inspection program, and confirmed its effectiveness by experiments.

Keywords: Computer viruses, Intrusion detection, Secure design / implementation

1. はじめに

本研究では，サイバー攻撃から情報システムを守ることを目的とし，その一つの手段として，IoT ボットの検知を効率的に行い，かつ誤検知が少なくなる方式としてボットの特徴的な自己の存在を隠蔽する挙動に基づいた検知方式を提案する．提案方式に基づいて検査プログラムを作成し，実際のマルウェアと正規プログラムを対象に実験を行い，提案方式が有用であることを確認した．

インターネットをはじめとした情報システムの発展は多種多様なサービスを生み出し，人々に多くの価値をもたらしてきた．しかしその一方で，その技術を悪用する者も存在し，不正アクセスや情報漏洩，迷惑メール，サービス妨害などのサイバー攻撃による被害は多発している．以前はこのような犯罪は自己顕示目的や愉快犯によるものが多かったが，現在ではビジネスとして不正な活動が行われている [1]．マルウェアをはじめとする電子的な攻撃はサービスとして売買されており，これらを防ぐ事は社会において非常に重要な課題である．

攻撃者は一般的にマルウェアを用いて攻撃を行うが，特に近年顕著なのがボット [2] による被害である．ボットとは，他人のコンピュータをマルウェアに感染させることで乗っ取り，そのコンピュータをネットワーク（インターネット）を通じて外部から操ることを目的として作成されたプログラムである．ボットは他人のコンピュータに感染すると外部からの指示を待ち，与えられた指示に従ってスパムメール送信活動やネットワーク感染活動などを実行する．

同一の指令サーバの配下にある複数のボットは，ボットネットと呼ばれる．攻撃者はボットネットを使って一斉に任意の対象に攻撃を仕掛けることが可能であるが，ボット

†1 現在，九州大学 工学部 電気情報工学科 計算機工学課程
Presently with Kyushu University, Fukuoka, 819-0385, Japan
†2 現在，九州大学 情報基盤研究開発センター
Presently with Kyushu University, Fukuoka, 819-0385, Japan
a) tajima.yuya.185@s.kyushu-u.ac.jp
b) koide@cc.kyushu-u.ac.jp

© 2020 Information Processing Society of Japan

はパソコンやスマートフォンなどの従来のインターネット接続機器だけでなく，現在爆発的に増加している IoT デバイス [3] にも感染することがあるため，非常に大きなネットワークを形成することができる．こうした事から，ボットは大規模で破壊的な攻撃が可能な凶悪なマルウェアであると言える．

さらに，ボットによる被害の拡大の理由として，ボットの総数の爆発的な増加がある．この増加は，ボットを始めとするマルウェアの亜種を簡単に作成できる環境が存在していることが原因と考えられる．攻撃者はマルウェアのソースコードや ZeuS[4] などと言ったマルウェア作成ツールキットを元に専門的な知識を持たずにマルウェアの亜種を作成することができ，特にツールキットは 1 つのボットから数百から数千の亜種ボットを作成できるようになっている．これらを利用して，攻撃者は攻撃を行っている．

ボットを始めとするマルウェアの検知の手法は，大きく 3 つに分類できる．3 つの検知手法の特徴を表 1 に示す．マルウェアの検知では一般にパターンマッチング [5], [6] と呼ばれる手法を用いて行われる．これはアンチウイルスソフトで多く採用されている手法であるが，事前にマルウェアを静的に解析してシグネチャを収集しておき，検査時に対象のファイルにそれが含まれるかどうかを静的に検査する方法である．この方法は既知のマルウェアに有効で，誤検知が起こりにくい手法であると言えるが，未知のマルウェアは既知のマルウェアに存在するシグネチャを有していないこともあるため検知することが難しい．先ほど述べた通り，亜種マルウェアが大量に生産されている現在では，パターンマッチングで検知できるマルウェアは一部に限られるのに加えて，暗号化処理など既知のマルウェアでもパターンマッチングを回避する手段があるため，パターンマッチングだけで全てのマルウェアを検知する事は非常に困難である．

未知のマルウェアを検知する手法としてパターンマッチングではなく，マルウェアの挙動に対するヒューリスティック法がある．事前にマルウェアを挙動という観点から解析，マルウェアらしい振る舞いとを定義し，検査時に定義した振る舞いと検査対象の振る舞いを比較する．ヒューリスティック法には静的に解析を行う静的ヒューリスティック法 [7], [8] と，動的に解析を行うビヘイビア法 (動的ヒューリスティック法)[9], [10], [11], [12], [13], [14], [15] の 2 つがある．

静的ヒューリスティック法は逆アセンブルにより生成したプログラムコードを解析し，対象がマルウェアらしい振る舞いをすると判断された場合，悪性と検知する．この手法ではあくまでプログラムコードによる振る舞いに着目しているため，亜種毎のバイナリの変化に影響されず，未知のマルウェアにも対応できる．しかし，マルウェアには解析回避のために難読化処理や暗号化処理を行われているものも存在し，そのようなマルウェアは静的ヒューリスティック法で検知するのには限界がある [16].

一方で，ビヘイビア法は静的ヒューリスティック法と同様に未知のマルウェアも検知できる他に，動的解析であるため難読化処理を施されたマルウェアも検知できるといった大きな利点があり，次世代のマルウェア検知手法として様々な研究が行われている．

ビヘイビア法の研究には大きく分けて 2 つのアプローチが存在する．1 つが機械学習による検知であり，もう 1 つがルールベースによる検知である．機械学習によるアプローチは様々な研究があり [11], [12], [13], [14], [15]，特に [15] では未知検体を最高で 96.8%の正解率でマルウェアであるかどうかを判定することに成功したことなどから，効果的なマルウェア検知手法であると考えられる．しかし，学習に大きなデータが必要なことや，新しいデータを追加学習させる場合，全データを再学習させる必要があるなど検知に必要なコストが高いことが非常に難点である．前述の通り未知ボットの多くは既存ボットの亜種であるため，一部のボットの特徴を抽出し，それをルールベースで検知するアプローチの方が IoT ボットの検知には適しているのではないかと考えられる．ルールベースによるアプローチにも誤検知が多いという問題がある．[10] はワームらしい挙動である自己複製挙動に着目した研究であるが，この自己複製挙動はマルウェアの感染のための主となる挙動であるが，正規プログラムにおいてもインストーラやアンインストーラで見られるため誤検知が発生してしまう．また，いくつかのボットは自らの感染や攻撃を円滑に行うために，自己の存在を隠蔽する挙動を行う．このような機能により，ボットは目立った挙動をあまり起こさないため，これまでに提案されてきた，被害に直接繋がるような危険な振る舞いを検知するビヘイビア型のマルウェア検知では感染に気づくことが困難になっていることもボットに対するビヘイビア法の共通の課題として挙げられる．

そこで本研究では，ボットの自己隠蔽挙動に着目し，これらをルールベースで検知する手法を提案する．自己隠蔽挙動を検知するため，前述の共通の課題を解決できる他，自己隠蔽挙動は正規プログラムでは起こりにくいと考えられるため，誤検知の削減も期待できる．自己隠蔽挙動に関する調査や研究は今までいくつか行われているが，それを逆用して対策に繋げるアプローチは比較的少ない．

ハニーポットとマルウェア配布サイトより入手した実際のボットの挙動を解析し得られた，自己の実行ファイルを削除する挙動と自己のプロセス名を変更する挙動，既存のコマンドを置き換える挙動の 3 つに着目し，それらをシステムコールを監視することで検知する検査プログラムを Linux 上で作成した．その後，この検査プログラムを用いて，既知のボットと正規プログラムに対して実験を行い，6 種類中 5 種類のボットを検知し，正規プログラムに関して

© 2020 Information Processing Society of Japan

は誤検知は発生しなかった．また検知できなかった 1 種類のボットに関しても，システムコール列には現れない自己隠蔽挙動を行っており，自己隠蔽挙動に基づく振る舞い検知という本提案方式においては検知できることが分かった．

以下，２章では本研究の位置づけと既存研究とその課題について述べる．３章では提案方式について述べ，４章で実装方法，５章では作成した検査プログラムを用いた検知実験と結果から得られた考察について述べる．最後に 6 章でまとめを行う．

2. 既存研究と本研究の位置づけ

マルウェアの検知を，その振る舞いに着目して行うビヘイビア法の研究には，マルウェアらしい振る舞いを事前に定義して検知を行うルールベースの検知と，マルウェアの振る舞いや正常プログラムの振る舞いの片方，もしくは両方を学習アルゴリズムによって学習し，検知を行う機械学習ベースの検知がある．本章では，それぞれの手法のこれまでの研究とその課題，そして本研究の位置づけについて述べる．

2.1 ルールベースによる検知

あらかじめ定義しておいたマルウェアらしい振る舞いを行ったプロセスを検知する取り組みの一つとして，マルウェアのセキュリティ無効化攻撃を逆用した検知 [9] がある．これはアンチウイルスソフトウェアやファイアウォールなどのセキュリティソフトウェアのプロセスを強制終了させるマルウェアの振る舞いを検知するものであり，いくつかの種類のマルウェアの検知に成功している．

また，ワームの拡散機能を逆用した検知 [10] では，ワームの，自己のファイルを読み，通信 API に出力するという動作のうちの自己のファイルを読むという挙動に着目し，検知を行っている．しかし，一部のアプリケーションのインストーラ，アンインストーラでは自己のファイルを読む挙動が観測されるため，誤検知の原因となる．

2.2 機械学習を用いた検知

機械学習を用いた検知の一つに，プロセスが発効したシステムコールの列の N-gram を用いる検知 [11], [12] がある．この検知において，N-gram とはあるプロセスによって実行されたシステムコール列の長さ N の部分列のことである．訓練モードにおいてシステムコールの N-gram を記録しながらアプリケーションを実行し，その記録を元にアプリケーションの正常な動作を表現する規則を作成する．そして，検査モードにおいて規則と照らし合わせながらアプリケーションを実行し異常を検知する．これは通常時のプロセスの振る舞いを定義する事で異常を検知する方式であるため，ルールベースの検知のように過去のマルウェアの特定の振る舞いに対する知識が必要でなく，また，簡単な定義であるためリアルタイムでの監視にも実装しやすいという利点がある．しかし，N の値が小さいため，長いスパンで現れるシステムコール間の関係を捉えられないという欠点がある．そこで，[11] では N の値を 5 と 6，11 として実験を行っていたが，後続の研究 [13] では，可変長の N-gram を用いる方式を提案しているものも存在する．また，別の方法として，システムコールシーケンスの解釈に有限オートマトンを用いる検知 [14] がある．この方法は N-gram では検知できないような長いシーケンスでの異常を検知することが可能である．上記の二つの検知方式は，アプリケーションごとに正常動作データが必要となるため，学習のコストが高いと言える．

機械学習を用いた未知マルウェアの検知には，既に非常に高い検知精度の結果を導き出している研究も存在する．[15] は同手法を用いて，未知検体を最高で 96.8%の正解率でマルウェア かどうか判定することに成功している．しかし現在一般的なバッチ処理型の機械学習は，新しいデータを追加学習させるには全データを再学習させる必要がある．マルウェアが大量に生産されている今日では，学習コストが高くなる可能性が高い．

2.3 本研究の位置づけ

機械学習を用いた検知方式は非常に効果的な検知手法であると言えるが，学習コストの面で問題がある．未知の IoT ボットの多くは既存のボットの亜種であり，その挙動は似ていることが多いため，現段階では学習コストのかかる機械学習による検知より，ルールベースによる検知の方が既存のシステムに実装しやすいため，この場合では優れていると考えられる．ルールベースによる検知では，問題点として主に誤検知の多さが考えられる．正規プログラムにおいては，特にインストールとアンインストールの挙動がマルウェアの挙動と似ているため，誤検知に繋がる．さらに両アプローチ共通の課題として，ボットは指令サーバから命令が送られてくるまで目立った挙動を起こさず，さらに自己の存在を隠蔽する処理を行うため、ビヘイビア法では検知しづらいことも挙げられる．

これらの課題を踏まえ，本研究では前述した自己の存在を隠蔽する挙動を逆用し，ルールベースで検知を行う方式を提案する．自己隠蔽挙動は正規プログラムでは現れないため，課題である誤検知も発生しないと考えられる．これらの隠蔽テクニックに関する調査はいくつか行われているが，それを逆用して対策に繋げるアプローチの研究はまだ少ない．

3. 提案方式

本章では，ボットの自己を隠蔽する挙動を逆用するボット検知手法のコンセプトについて述べる．

© 2020 Information Processing Society of Japan

表 1　マルウェア検知手法.
Table 1　Malware detection method.

手法	検知方法	未知マルウェア	暗号化マルウェア
パターンマッチング法	静的なシグネチャ検知	×	○
静的ヒューリスティック法	静的な振る舞い検知	○	×
ビヘイビア法	動的な振る舞い検知	○	○

3.1　自己隠蔽挙動

ボットは感染時，以降の不正な活動を円滑に行うために自己の存在や痕跡を削除，隠蔽することが多い．具体的な手法として，自己の実行ファイルを削除する挙動（自己ファイル削除）と自己のプロセス名を変更する挙動（プロセス名変更），および netstat や ps コマンドといった不正な活動を突き止めるための第一手段となるようなコマンドの実行ファイルを別のプログラムの実行ファイルに置き換える挙動（既存コマンド置き換え）がある．これらは，ハニーポットとマルウェア配布サイトより入手した実際のボットを仮想環境上で，プログラムが使用するシステムコールと受け取るシグナルを監視することのできる strace をという Linux のデバッグユーティリティを用いて実行し，出力されたシステムコール 列を解析することで得られた．本提案手法では，自己隠蔽挙動として定義された，自己ファイル削除，プロセス名変更，既存コマンド置き換えの 3 つの挙動を検知する．以下で自己隠蔽挙動のそれぞれの説明を述べる．

3.1.1　自己ファイル削除

これは Mirai[17] を解析することで得られた特徴である．ボットを始めとするマルウェアの感染は，実行ファイルを対象コンピュータにダウンロードさせ，自動実行もしくはユーザにより実行で開始する．Mirai は検知や解析を困難にするために，実行直後に自分自身の実行ファイルを自ら削除する挙動を行う．これらのボットらは，自己ファイル削除のために，unlink（"{ 自己ファイルの path}"）システムコールを使用していた．実行ファイルを削除しても，プログラム自体は既にメモリにロードされているため，実行中のボットには何も影響は出ないが，コンピュータの電源を落としてしまうと，基本的にプロセスも終了するため Mirai は活動できなくなり，感染の証拠も失われる．unlink システムコール自体は，正規プログラムにおいて非常に高頻度で発行されるが，そのプロセスの発行元を自ら削除することは，悪性プログラム特有のものであると仮定し，この自己ファイル削除という挙動をボットらしい自己隠蔽挙動として定義づけた．

3.1.2　プロセス名変更

これは Mirai を解析することで得られた特徴である．ps コマンドなどで現在動作しているプロセスを表示する事で，不審なプログラムが動作している事に気づくことが可能なことは多い．それを回避するために Mirai はプロセス名を prctl（PR_SET_NAME，"{ ランダムな文字列 }"）を使用して変更する．プロセス名変更を行う事で不正なプロセスの活動を隠す事や，そのプロセスの元となるファイルの存在を隠す目的があると考えられる．この挙動は，正規プログラムでは起こり得ない挙動であるため，誤検知に繋がらないと考えられるためボットらしい自己隠蔽挙動として定義づけた．

3.1.3　既存コマンド置き換え

3.1.2 項のプロセス名変更では不正なプロセスの名前を変更して表示させる挙動であったが，そもそも不正なプロセスなどを表示させないようにする挙動が BillGates Linux Botnet と呼ばれるボットで見られた．具体的には，netstat, lsof, ps, ss のコマンドによって実行されるファイルを自身の実行ファイルに置き換える事で，それらのコマンドから不正な活動が露見することを妨害していた．BillGates Linux Botnet が置き換える実行ファイルのリストを表 2 に示す．今回はこの中で，/bin/netstat, /bin/lsof, /bin/ps, /bin/ss を置き換えが起こった際に検知する．置き換えには BillGates Linux Botnet は cp を用いて行なっているが，本研究では cp を実行する際に呼び出される，unlink（"{ コピー先 path}"）をボットらしい自己隠蔽挙動として定義づけた．これらの挙動は正規プログラムでは起こり得ない挙動であるため，誤検知につながらないと考えられる．

3.2　検知方法

3.1 項で述べたように，ボットは侵入活動で自己の存在を隠蔽するために挙動が見られることがあり，この挙動を逆用することで検知することが可能であると考えられる．また，ボットは一つのソースコードから大量に亜種が生産されるため，自己隠蔽挙動は亜種間で引き継がれる可能性があることも考えられる．そこで，ボットのシステムコールを監視し，事前に定義したシステムコールを行なったプロセスを悪性だと判断する検知手法を提案する．判断基準となるシステムコールの種類とその引数を表 3 に，検知のフローチャートを図 1 に示す．

システムコールの監視にはシステムコールフックを用いた．フックの具体的な手法は次章の実装で述べる．

4.　実装

提案方式の有用性を評価するため，IoT ボットの自己隠

表 2　BillGates Linux Botnet が置き換えるコマンド．
Table 2　Commands that BillGates Linux Botnet replaces.

実行ファイル	動作
/bin/netstat	ネットワーク接続状態やソケット/インターフェイスごとのネットワーク統計などの表示
/bin/lsof	オープンしているファイルの一覧の表示
/bin/ps	実行されているプロセスの一覧の表示
/bin/ss	ネットワーク通信で利用するソケットの情報などの表示
/usr/bin/netstat	/bin/netstat と同じ
/usr/bin/lsof	/bin/lsof と同じ
/usr/bin/ps	/bin/ps と同じ
/usr/bin/ss	/bin/ss と同じ
/usr/sbin/netstat	/bin/netstat と同じ
/usr/sbin/lsof	/bin/lsof と同じ
/usr/sbin/ps	/bin/ps と同じ
/usr/sbin/ss	/bin/ss と同じ

表 3　検知対象となるシステムコールと引数．
Table 3　System call and arguments to be detected.

システムコール名	第 1 引数
unlink	自己ファイルの path
prctl	PR_SET_NAME
unlink	/bin/netstat /bin/lsof /bin/ps /bin/ss

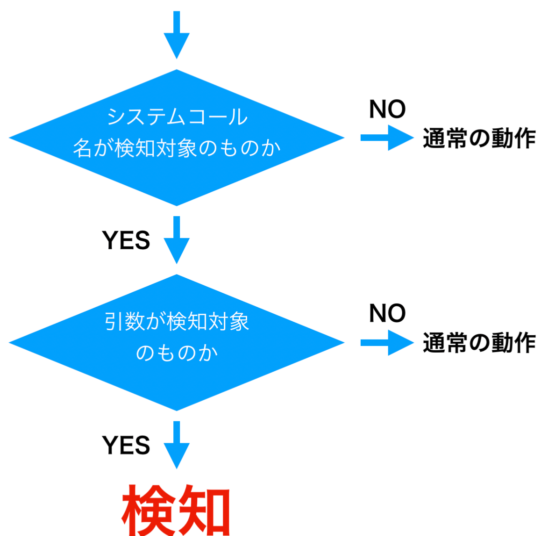

図 1　検知のフローチャート．
Fig. 1　The flowchart of detection.

蔽挙動システムコールレベルで検知する，自己隠蔽挙動検査プログラムを作成した．本実装では，OS には Alpine inux 3.5，カーネルに Linux-4.4 カーネル，アーキテクチャに i686 アーキテクチャを使用した．本章ではその実装方法について述べる．

4.1　実装の概要

システムコールフックを用いて提案方式を実装した．システムコールフックとはシステムコール発行時に本来想定された処理ではなく，別の処理を行うことである．本実装では，unlink と prctl の 2 つのシステムコールをフックし，フック関数に遷移させ，そこで引数のチェックを行うことで提案方式を表現した．以下で，システムコールフックの概要，本実装のシステムコールフックの詳細，フック関数の処理について詳しく述べる．

4.2　システムコールフック

本節では，通常のシステムコールの流れとシステムコールフックの概要についての説明を述べる．システムコールとは，OS がユーザモードプロセスに提供する CPU，ディスク，プリンタなどのハードウェアとのやり取りをするインターフェイスことである．通常，Linux/x86 ではシステムコールはレジスタにシステムコール番号と引数を格納し発行するが，その際のレジスタ設定は C 言語では記述できない．これらはアセンブラで記述されるが，その作業をプログラマに任せるのは非効率的なため，C 言語から呼び出すことができる関数の形にライブラリ化して，システム側から提供されている．そのため C 言語で，例えば write() を呼び出せば，ライブラリ側でレジスタの処理とシステムコールの発行が行われる．このような処理を行うアセンブラで書かれた関数はシステムコール・ラッパと呼ばれている．システムコールが発行されるとソフトウェア割り込みが発生し，CPU は Linux カーネルの割り込み処理の実行に移る．その後システムコールハンドラが呼び出され，その中でシステムコール番号と合致するシステムコール関数のポインタを sys_call_table から見つけ出し，そのポインタにジャンプする．ジャンプ後，システムコールが引数を取り出し作動する．

本研究のシステムコールフックは，この sys_call_table

に登録されているシステムコール関数のポインタをフック関数のポインタに書き換えることでシステムコールが発行された時，その処理を横取り（フック）することを実現している．

4.3 本実装でのシステムコールフック

4.2 で述べた通り，sys_call_table の一部のシステムコール関数のポインタを書き換えることでシステムコールフックを行った．実際には，表 3 で示されている検知対象となるシステムコール unlink，prctl のシステムコール関数のポインタをフック関数のものに書き換えた．図 2 は unlink をフックする際の sys_call_table の書き換えを示す．また，この時元のシステムコール関数のアドレスは別の領域に退避させた．書き換えのために sys_call_table のアドレスを取得しなければならないが，Linux カーネル 2.6 以降はセキュリティ上の理由から sys_call_table のシンボルがエクスポートできない．そのため本実装ではシェル上で cat /proc/kallsyms | grep sys_call_table を実行し，表示されたアドレスを導入のためのコードに直接ハードコーディングすることでこの問題を解決した．

本実装では，sys_call_table の書き換えを LKM（Loadable Kernel Module）を使用して行なった．LKM とは OS の動作中のカーネルを全体の再構築なしで拡張するためのオブジェクトファイルのことである．

システムコールフックを開始すると，全てのユーザプログラムがシステムコールを使用する際に書き換えられた sys_call_table をシステムコールハンドラを介して参照し，フックされるシステムコールだった場合，フック関数に処理を移す．図 3 はユーザプログラムが unlink（システムコール番号：10）を発行した時のフックの概要を示す．

4.3.1 フック関数

本節ではフック関数内部の処理についての説明を行う．フック関数中では，引数を検査し，悪性だと判断できる場合はプロセス ID とそのプロセスの振る舞いをログに出力することで検知とした．

4.3.1.1 unlink のフック関数

unlink は第 1 引数に削除するファイルの path が設定される．この第 1 引数を検査することによってマルウェアかどうかを判断する．まず自己ファイル削除であるかの検査を行う．Linux のプロセスは task_struct 構造体というデータ構造体で表現されており，このタスク構造体の中にはプロセス ID やプロセスのコマンド名などが格納されている．本実装では unlink の第 1 引数が自己ファイルであるかの判断を，このタスク構造体のコマンドと文字列比較することで行ない，一致した場合は自己ファイル削除であるとして検知するよう作成した．タスク構造体のコマンド名は最大で 16 文字までしか格納できないため，第 1 引数が 16 文字以上の場合は，第 1 引数の 16 文字目までがコマンド名と一致していた時，自己ファイル削除として検知するよう作成した．

次に既存コマンド置き換えであるかの検査を行う．こちらも先ほどと同様に，第 1 引数と /bin/netstat, /bin/lsof, /bin/ps, /bin/ss を文字列比較し，一致すれば既存コマンド置き換えであるとして検知するよう作成した．

4.3.1.2 prctl のフック関数

prctl は第 1 引数にオプションを設定することでどのような動作を行うか決定する．プロセス名変更を行うオプションは PR_SET_NAME である．これらオプションは整数型で指定するが，一般にそれらは sys/prctl.h において定義されている．本環境では PR_SET_NAME は 15 であった．したがってフック関数では第 1 引数と 15 を比較し一致するならばプロセス名変更として検知するよう作成した．

本実装では，マルウェアであると判断してもそのプロセスの活動を停止させず，あくまで検知だけを行い，検査終了後は従来の unlink と prctl の関数に処理を渡してプログラムが想定した動作を行わせるよう実装した．

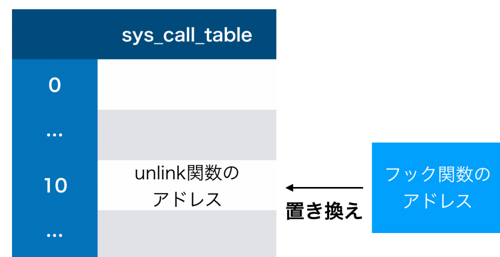

図 2 フック関数のロード．
Fig. 2 Load of hook.

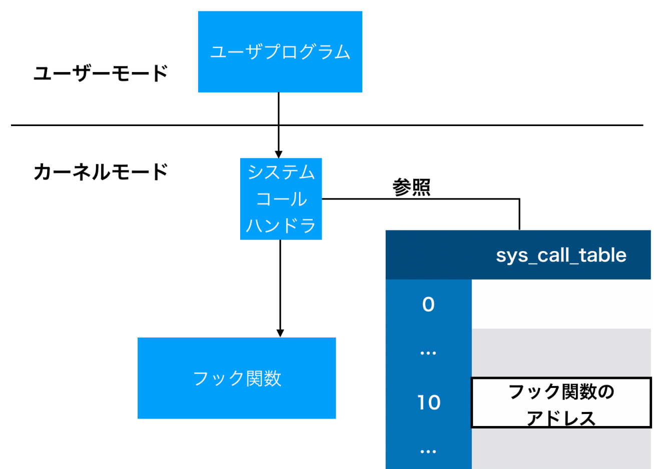

図 3 フックの概要．
Fig. 3 Overview of hook.

5. 実験と考察

本章では，検体を用いた検査プログラムの検知実験と正

規プログラムを用いた誤検知実験の方法と結果，そして考察を行う．

5.1 検知実験

5.1.1 実験方法

提案方式の有用性を検証するため，実際のボットを用いて検査プログラムによる検知実験を行なった．4 章において作成したカーネルモジュールをロードし，root 権限で検体を実行した．

検体は，ハニーポット Cowrie[18] にて独自に収集したものとマルウェア配布サイトより入手した 6 体を使用した．実験環境は仮想マシン Oracle VM Virtual Box 上の Alpine Linux 3.5 で，ネットワーク環境はホストオンリーにて行なった．

5.1.2 実験結果

上記の実験を行なった．その結果を表 4 に示す．○は検知を表し，×は検知できなかったことを表す．一部の検体は，本環境がインターネット接続ができないため，自己隠蔽挙動を検査できなかった．そこで検体のバイナリを逆アセンブルし解析することで，インターネット接続できる場合の処理を確認し，本提案手法の実装で検知できるかの調査も同時に行った．静的検査を行い，本提案手法で検知できると分かったものは (○) と表し，検知できないと分かったものは（×）と記述する．

表 4 検査プログラムによる検知実験の結果．
Table 4 Results of detection experiment by inspection program.

ボット	結果	検知理由
Mirai	○	自己ファイル削除 プロセス名変更
Gafgyt	○	プロセス名変更
Tsunami	×（×）	
Ganiw	○	既存コマンド置き換え
BillGates	○	既存コマンド置き換え
XOR DDOS	○	自己ファイル削除

実験の結果，5 種類のボットに対して，検知することが成功した．

5.2 誤検知実験

5.2.1 実験方法

ボットの検知実験と同様に，正規プログラムを提案方式下で実行し，誤検知が発生するかの調査を実験にて行なった．本実験では Alpine Linux のパッケージ管理ツール apk を用いた一般的なアプリケーションのインストールとアンインストールを対象に行なった．

5.2.2 実験結果

結果を表 5 に示す．○は誤検知を表し，×は誤検知しなかったことを表す．

表 5 検査プログラムによる誤検知実験の結果．
Table 5 Results of false-detection experiment by inspection program.

正規プログラム	結果
wireshark（インストール）	×
wireshark（アンインストール）	×
git（インストール）	×
git（アンインストール）	×
mysql（インストール）	×
mysql（アンインストール）	×
vim（インストール）	×
vim（アンインストール）	×

以上の結果より，誤検知実験では提案方式における正規プログラムの誤検知は見られなかった．

5.3 考察

5.1 項の検知実験より，一定数のボットに対して提案方式は有効であると考えられる．また 5.2 項の誤検知実験より，今回実験を行なった正規プログラムにおいては誤検知は発生しなかった．以上のことから，本提案手法はビヘイビア型のマルウェア検知の課題である誤検知の発生を解決しつつ，一定数のボットを検知できるため，有用であると考えられる．

5.1 項の検知実験で検知できなかった Tsunami は，静的解析を行ったところ，本提案手法で自己隠蔽挙動として定義した挙動は行わなかったが，別の自己隠蔽挙動を行なっていることがわかった．具体的には，プログラムの実行ファイルの path であるコマンドライン引数の 0 番目を usr/sbin/sshd に書き換えることがわかった．これにより，ps コマンドの表示が変わり，不正な活動が発見されにくくなる．以上の処理はプロセス内のメモリの読み書きのみであるため，システムコールを使用しない．したがって今回の検査プログラムでは検知できなかったが，自己隠蔽挙動自体は確認できたため，検査プログラムを拡張することで，本提案方式で検知できるようになると考えられる．

6. おわりに

6.1 本研究の主たる成果

本論文では，ボットの自己隠蔽挙動に着目し，自己隠蔽挙動を検知することで IoT ボットの検知を効率的に行い，かつ誤検知が少なくなるような検知方式を提案した．LKM を用いてシステムコールテーブルを書き換えて実現したシステムコールフック用いて提案方式を実装し，自己隠蔽挙動を検知する検査プログラムを作成した．提案方式が有効であるかを確認するために，仮想環境上でボットを実行し，検査プログラムを用いて検知実験を行なった．さらに，いくつかの正規プログラムを実行し，検査プログラムによる誤検知が発生しないかの実験を行なった．実験の結果，5

© 2020 Information Processing Society of Japan

体のボットを検知し，また誤検知は一度も発生しなかった他，実験において検知できなかった検体においても，本実装方法では検知できない自己隠蔽挙動を行なっていたため，自己隠蔽挙動に基づいた検知という本提案方式は有用であることが確認された．

6.2 今後の課題

本研究では，3 つの自己隠蔽挙動に着目して検知を行ない，自己隠蔽挙動による検知は有用であることが分かった．しかしシステムコールの監視だけでは検知できないものも存在するなど，自己隠蔽挙動は他にも様々なものが存在すると考えられる．今後は他の自己隠蔽挙動を検知できるよう提案方式や実装方法を改善していく必要がある．また，本提案方式は，検知時悪性プロセスのプロセス ID とその振る舞いをログに出力するが，悪性プログラムは複数プロセスにわたって存在することが多く，ログに記述された情報だけでは脅威を取り除くことができない．今後は，検知時に脅威を完全に取り除くようなシステムを追加し，改善していく必要がある．

謝辞 本研究の一部は日立システムズの援助を得ている．

参考文献

[1] Nathan Friess and John Aycock: *Black market botnets*, (2007).
[2] 独立行政法人情報処理推進機構 セキュリティセンター：ボット対策について，入手先 ⟨https://www.ipa.go.jp/security/antivirus/bot.html⟩ (参照 2020-01-27)
[3] 総務省：爆発的に増加する IoT デバイス，入手先 ⟨https://www.soumu.go.jp/johotsusintokei/whitepaper/ja/h29/html/nc133100.html⟩ （参照 2020-01-27）
[4] Nicolas Falliere and Eric Chien: *Zeus: King of the bots*, Symantec Security Response (http://bit. ly/3VyFV1) (2009).
[5] Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt: *Fast pattern matching in strings*, SIAM journal on computing, Vol.6, No.2, pp.323-350 (1997).
[6] Christopher Kruegel and Thomas Toth: *Using decision trees to improve signature-based intrusion detection*, International Workshop on Recent Advances in Intrusion Detection, pp.173-191 (2003).
[7] Mihai Christodorescu, Somesh Jha, Sanjit A Seshia, Dawn Song, and Randal E Bryant: *Semantics-aware malware detection*, 2005 IEEE Symposium on Security and Privacy (S&P'05), pp.32-46 (2005).
[8] Jean Bergeron, Mourad Debbabi, Mourad M Erhioui, and Béchir Ktari: *Static analysis of binary code to isolate malicious behaviors*, Proceedings. IEEE 8th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE'99), pp.184-189 (1999).
[9] 松木隆宏, 新井悠, 寺田真敏, 土居範久, et al.: セキュリティ無効化攻撃を利用したマルウェアの検知と活動抑止手法の提案, 情報処理学会論文誌, Vol.50, No.9, pp.2127-2136 (2009).
[10] 松本隆明, 鈴木功一, 高見知寛, 馬場達也, 前田秀介, 水野忠則, 西垣正勝, et al.: 自己ファイル *READ* の検出による未知ワームの検知方式, 情報処理学会論文誌, Vol.48, No.9, pp.3174-3182 (2007).
[11] Stephanie Forrest, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff: *A sense of self for unix processes*, Proceedings 1996 IEEE Symposium on Security and Privacy, pp.120-128 (1996).
[12] Steven A Hofmeyr, Stephanie Forrest, and Anil Somayaji: *Intrusion detection using sequences of system calls*, Journal of computer security, Vol.6, No.3, pp.151-180 (1998).
[13] Carla Marceau: *Characterizing the behavior of a program using multiple-length n-grams*, Proceedings of the 2000 workshop on New security paradigms, pp.101-110 (2001).
[14] R Sekar, Mugdha Bendre, Dinakar Dhurjati, and Pradeep Bollineni: *A fast automaton-based method for detecting anomalous program behaviors*, Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001, pp.144-155 (2000).
[15] Ivan Firdausi, Alva Erwin, Anto Satriyo Nugroho, et al: *Analysis of machine learning techniques used in behavior-based malware detection*, 2010 second international conference on advances in computing, control, and telecommunication technologies, pp.201-203 (2010).
[16] Andreas Moser, Christopher Kruegel, and Engin Kirda: *Limits of static analysis for malware detection*, Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), pp.421-430 (2007).
[17] The New Jersey Cybersecurity and Communications Integration Cell (NJCCIC): *Mirai Botnet*, 入手先 ⟨https://www.cyber.nj.gov/threat-profiles/botnet-variants/mirai-botnet⟩ （参照 2020-01-28）
[18] Cowrie, リポジトリ, 入手先 ⟨https://github.com/cowrie/cowrie⟩ （参照 2020-02-05）

© 2020 Information Processing Society of Japan