

# RDMA を用いたメモリ監視によるマルウェア検知に向けたシステムコールログ削減手法

中野 峻作<sup>1</sup> 渡邊 和樹<sup>2</sup> 佐々木 一樹<sup>2</sup> 鷲尾 元太郎<sup>2</sup> 穂山 空道<sup>1,a)</sup>

**概要：**データセンタ内をラテラルムーブメントするマルウェアの検知には、データセンタ全体の監視が必要である。データセンタ全体を監視する手法として各マシンに監視用エージェントを配置し侵入検知システムの動作するマシンに通知する方法が考えられるが、マルウェアによるエージェントの停止・改竄や監視処理の負荷による検知回避の危険性がある。そこで本研究では RDMA により単一の監視マシンからデータセンタ内の複数マシン上のメモリダンプを取得し、取得したメモリダンプからマルウェアを検知することを提案する。監視対象マシンの数が増加した際にも提案手法が有効に働くようにするため、2 種類のデータ転送量削減手法を提案する。提案手法をサーバ環境を模した 4 つのケースで評価し、監視マシンと監視対象マシン間のデータ転送量を約 7.9 倍削減できることが分かった。

## 1. はじめに

データセンタ内をラテラルムーブメントするマルウェアが脅威となっており、その検知にはデータセンタ全体の監視が必要である。ラテラルムーブメントとは攻撃者がネットワークへ侵入した後、ネットワーク内部を移動しながら侵害範囲を段階的に拡大する攻撃手法である。ラテラルムーブメントの被害の例として 2025 年のアサヒグループの事例が挙げられる。この事例ではマルウェアがデータセンタ内の一台のサーバに侵入した後、その他のマシンへ感染を拡大させた [1]。このような攻撃ではマルウェアが短時間で複数マシン間を移動するため、各マシンでの検知ではタイミングによって攻撃を見逃す可能性があり、データセンタ全体での監視が必要である。

ラテラルムーブメントに対する従来手法として各マシン上のエージェントで自マシンを監視する方法が考えられるが、この方法ではマルウェアによる監視機構の改竄・停止の可能性がある。例えば脆弱性のあるドライバを標的組織の環境内に持ち込み不正利用を行う BYOVD (Bring Your Own Vulnerable Driver) 攻撃では監視エージェントを停止・改竄しうる。具体的にはランサムウェアの Kasseika は BYOVD 攻撃を利用してアンチウイルスのプロセスやサービスを強制停止することで検知を回避した [2]。

そこで本研究では 1 台の監視マシンからデータセンタ内

の複数のマシンを監視しマルウェアを検知する手法を提案する。本手法では監視マシンから RDMA で監視対象マシンのメモリダンプを取得し、取得したメモリダンプからマルウェアに関する情報を抽出する。RDMA による監視は、次の三つの理由によりマルウェアによる改竄・停止を防止できる。

- (1) 監視機構をマルウェアが実行されている監視対象マシン外に設置することで、マルウェアによる改竄・停止が困難になる。
- (2) 監視対象マシンが能動的に通信する必要がなくなることで、マルウェアが監視されていることを検知できなくなる。
- (3) 監視対象マシンのオーバーヘッド削減によってマルウェアが監視されていることを検知できなくなる。

本手法の実現には二点の技術的課題がある。一点目は監視マシンが受信するデータ量が膨大になる点である。マルウェア検知に必要な情報は単一マシンからでも大量に生成され、複数のマシンを一括で監視すると監視マシンが受信するデータ量はさらに増大する。二点目はマルウェアに関する情報の抽出が困難である点である。RDMA で取得できる情報はメモリダンプであり、そこからマルウェアに関する情報を抽出するにはセマンティックギャップの解消が必要である。本研究では二点目の課題は監視対象マシン上で実行中のプロセスのシステムコール呼び出し履歴を収集するプロセスを利用することで代用し、一点目の課題の解決に注目する。

受信データ量の課題を解決するため、本研究ではプロセ

<sup>1</sup> 立命館大学 情報理工学部

<sup>2</sup> 三菱電機株式会社 情報技術総合研究所

a) s-akym@fc.ritsumeit.ac.jp

スペースとシステムコールベースの二種類のデータ量削減手法を提案する。プロセスベースのデータ量削減手法はシステム起動時に存在する正規プロセスを監視対象外にする。システムコールベースのデータ量削減手法ではマルウェアによって頻繁に悪用されるシステムコールの呼び出し履歴のみを取得する。

提案手法の有効性を評価するために、サーバ環境を想定しデータ量削減の評価を行った。具体的には監視対象マシン上でマルウェアを実行し、RDMA による監視を行った際のデータ転送量を提案手法の有無で比較した。評価の結果、提案手法によって転送すべきデータ量を約 7.9 倍削減することに成功した。

## 2. 研究背景

### 2.1 ラテラルムーブメント

ラテラルムーブメントとはマルウェアが組織の内部ネットワークの侵入に成功した後、ネットワーク内を横移動し侵害範囲を拡大する攻撃手法である。攻撃者はマルウェアによって組織の内部ネットワークに侵入すると、同一ネットワーク上の複数のマシンに感染を拡大させる [3]。

近年のラテラルムーブメントの被害事例として 2025 年に発生したアサヒグループのランサムウェア被害が挙げられる。この事例では初期侵入としてグループ拠点のネットワーク機器へ侵入し、権限奪取と横展開によってデータセンタへと侵入した [1]。

ラテラルムーブメントを検知するには複数マシンのログを集約・相関することが有効である。ラテラルムーブメントは複数のマシンにまたがって実行されるため、単一マシンにおける監視・検知のみではマルウェアの侵入元の特定制が困難であり、攻撃の全体像が把握できない [4]。

### 2.2 マルウェアの検知手法

#### 2.2.1 ヒューリスティック検知

ヒューリスティック検知とはプログラムの中身、挙動を元にマルウェアを検知する手法である。この検知手法ではマルウェアの特徴的な挙動の有無を調べることによって検知を行う。これによってマルウェア固有のパターンを照合するシグネチャベースの検知手法では検知が困難な未知のマルウェアを検知できる。ヒューリスティック検知は静的ヒューリスティック検知と動的ヒューリスティック検知の 2 種類に分けることができる。

静的ヒューリスティック検知はプログラムのソースコードを検証し、既知のマルウェアと比較することでマルウェアを検知する手法である。この手法ではプログラムを実際に動かすことなくソースコードからプログラムを動作させた場合の振る舞いを予測して検知を行う。

動的ヒューリスティック検知はプログラムを実行して挙動を見て検知する手法である。また動的ヒューリスティッ

ク検知ではプログラムを実行する際、安全のためにサンドボックスと呼ばれる仮想環境を構築し、その中で対象のプログラムを実行する場合もある。プログラム実行中のシステムコール呼び出しなどを監視することでマルウェアを検知する。静的ヒューリスティック解析では難読化されたプログラムを検知することが困難であるのに対し、動的ヒューリスティックではプログラムを実際に動かして検知するため、静的ヒューリスティック検知で検知できないマルウェアを検知できる。しかし静的ヒューリスティック検知はプログラムのソースコードを見るだけなのに対し、動的ヒューリスティック検知ではマルウェアを実際に動かすため判定に時間がかかり、システムに負荷がかかるという課題がある。またデバイス情報などを参照することによりサンドボックスのような仮想マシンで実行されていると判断した場合、不正な挙動を止めて正体を隠すマルウェアも存在するため [5]、このようなマルウェアを検知するのは困難である。

#### 2.2.2 システムコール

システムコールのトレースによってマルウェアを検出することが可能である。マルウェアがファイルの改竄や暗号化、ネットワーク通信によって悪意のある操作を行う場合、必ずシステムコールを介して処理を行う必要がある。よってこれらの挙動をシステムコールのログから観察することでマルウェアを検知できる。

#### 2.2.3 メモリフォレンジック

メモリフォレンジックとはメインメモリ上のデータを解析し、コンピュータ上でプログラムがどのように動作していたかなどの情報を得ることで、不正行為やセキュリティインシデントの証拠を収集する技術である。メモリフォレンジックは以下のような手順で行われる。

(1) メモリダンプの取得

(2) メモリ解析

メモリフォレンジックはメインメモリの内容をコピーしてストレージにメモリダンプとして保存することから始まる。メモリダンプの取得が完了したら、メモリ解析を行いメモリ内に格納されている実行中プロセス、アクティブなネットワーク接続、読み込まれているカーネルモジュールなどに関する情報を取得し、その情報をもとにインシデントレスポンス、マルウェア検知を行う。

ファイルレスマルウェアはストレージに痕跡を残さずメモリ上でのみ活動するため、メモリに格納されている情報を直接解析するメモリフォレンジックはこの手法に対して有効である。また一部のマルウェアはシステムコールなどの動作を改変するフッキングや OS の設定を変更することなどにより検出を回避するが、これらの回避手法は OS が提供する監視システムやその他の監視ツールの検知を妨害するものであり、メモリ上にマルウェアの実行コード、ネットワーク接続に関する情報は残る。よってメモリフォ

レンジックを用いることでこれらの検出回避手法の影響を受けない。

## 2.3 RDMA

Remote Direct Memory Access(RDMA)とは、あるコンピュータから遠隔のコンピュータのメモリへOS、CPUを介さずにアクセスできる技術である。RDMAではCPUを介さず通信できるため、従来の通信手法よりも遠隔のコンピュータと高速な通信が可能である。またCPUの使用率を削減できるため、コンピュータ上で動作する他の処理にリソースを割り当てられるという利点がある。RDMAを用いたデータ転送の概要を図1に示す。従来のネットワーク通信ではCPUを介してメモリにアクセスすることでデータのやり取りを行うが、RDMAではCPUをバイパスしてNIC(Network Interface Card)から直接メモリにアクセスする。

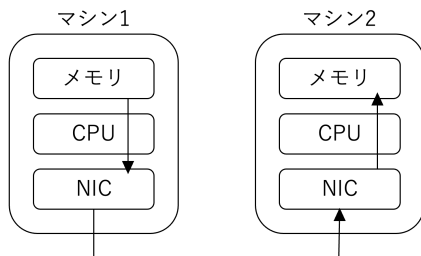


図1: RDMAを用いたデータ転送の流れ

RDMAを用いたデータの送受信ではRDMAに対応したネットワークインターフェースカード(RNIC)がメインメモリ上のデータにDirect Memory Access(DMA)を行うことで実現される。この際、RNICはユーザープロセスの仮想アドレスを物理アドレスに変換することができないため、RDMAではMemory Region(MR)という領域をあらかじめOSに登録する。MRの登録により、対象領域の仮想アドレスと物理アドレスの対応が固定化され、その対応表をRNICに渡すことでRNICから直接アクセスすることが可能となる。

## 3. 関連研究

### 3.1 メモリフォレンジックを用いたマルウェアの自動検出

Nairら[6]は、メモリフォレンジックによるマルウェア検出を自動化するコマンドラインツールを提案した。このツールではメモリダンプから疑わしいプロセスを特定し、抽出したプロセスのバイナリをVirusTotalを用いて検証する。疑わしいプロセスの特定にはメモリ解析ツールであるVolatility3を用いてアーティファクトの抽出、分析を行う。具体的にはコマンドラインの実行履歴、プロセスのリスト、ネットワーク接続の詳細を抽出する。

本ツールは監視対象マシンの上で能動的に動作するもの

であり、そのままではデータセンタ全体を一括で監視できない。これに対し本研究ではRDMAを用いてデータセンタ全体を監視しラテラルムーブメントするマルウェアを追跡・検知する。

### 3.2 メモリダンプ取得時の負荷

メモリダンプからのマルウェアを検知にはメモリダンプ取得の負荷が大きいという課題が一般にある。メモリダンプ取得の負荷が大きいことは(1)大量のメモリやストレージIOによるシステムの別のプロセスの性能に影響を及ぼす可能性がある、(2)取得に時間がかかるため取得中にメモリ内容が変化しマルウェアに関する重要な痕跡を削除してしまう可能性があるという課題がある。

メモリダンプ取得時の負荷の具体例として、Oliveriらの研究ではLinuxのメモリダンプツールであるLiMEとMicrosoft AVMLにおけるメモリダンプ取得による影響を示している[7]。LiMEはカーネルモジュールを用いてカーネル空間からメモリダンプを作成するツールであり、メモリダンプ作成中にカーネルが55億7600万回のストレージへの書き込み操作を実行し、573861個の物理ページと合計38GiBのデータが新たに書き込み、上書きされることが報告された。Microsoft AVMLはユーザー空間から/proc/kcoreを使ってメモリダンプを作成するツールであり、AVMLの実行中には9億7100万回のストレージへの書き込み操作を実行し、796709個の異なる物理ページと合計68GiBのデータが書き込みされることが報告された。

メモリダンプ取得時の負荷に対し、本研究では二つの方向で解決を図る。一点目はRDMAを用いることでメモリIO以外の負荷(監視対象マシンでのCPU処理、ストレージIO)を軽減すること、二点目は取得すべきデータ量を厳選し負荷を削減することである。

### 3.3 RDMAを用いたメモリイントロスペクション

Liuら[8]は仮想化環境においてハイパーバイザから仮想マシン内を監視する際の負荷を削減するため、RDMAとP4プログラマブルスイッチを用いて仮想マシンの監視を実現した。具体的には各ラックに存在するP4プログラマブルスイッチが各マシンのメモリ内容をRDMA経由で取得することで仮想マシン内の不審な活動を監視する。メモリ内容の解析処理はP4プログラマブルスイッチ内のASICで行うことで高速の処理を行う。この手法によって6種類のルートキットの検出に成功し、ハイパーバイザーベースのメモリイントロスペクションツールであるLibVMIで1.3~6.2個のCPUコアが必要なイントロスペクションタスクをCPUオーバーヘッドゼロで行うことが可能になった。

本手法では仮想マシン全体を低負荷に監視できる一方、各ラックにP4プログラマブルスイッチの存在を仮定する。

これに対し本手法では監視マシンはデータセンタ内の通常のノードであり、またマルウェアの検知のみを対象とすることで高性能な ASIC を必要としない。

#### 4. ラテラルムーブメントを行うマルウェアの検知の課題

ラテラルムーブメントを行うマルウェアを検知にあたり、次の三つの課題がある。

- (1) 単一ホストでの監視におけるマルウェアの追跡困難性
- (2) マルウェアによる監視機構の改竄・停止
- (3) 監視処理の高負荷による検知回避リスク

ラテラルムーブメントではマルウェアが内部ネットワークに侵入したあと、同一ネットワークのマシンを横断する。各マシン上にエージェントを設置し、監視・検知を行う場合、マルウェアによる攻撃の経路を追跡することができず、攻撃の全体像を把握することができない。

監視機構がマルウェアが実行されているマシン上に設置されている場合、マルウェアはBYOVD 攻撃などを利用して監視機構を改竄・停止する可能性がある。2023 年に発見されたランサムウェア Kasseika は BYOVD を利用してアンチウイルスのプロセスやサービスを強制停止することで検知を回避した [2]。

マルウェアは監視機構の負荷から自身が監視、解析されていることを検知し、正体を隠す可能性がある [9]。例としてメモリフォレンジックを用いてマルウェアを検知する手法ではメモリダンプ取得時にシステムに大きな負荷がかかり、この負荷がマルウェアが正体を隠す原因となる。

## 5. 提案手法

## 5.1 脅威モデル

本研究ではマルウェアはユーザ空間で動作し、攻撃者が持つ C&C サーバとの通信やファイル操作が可能であると仮定する。一方で管理者権限が必要な操作はできないと仮定する。例えば RDMA の通信に用いるカーネル内バッファにアクセスし改竄することはできない。またマルウェアがシステムの正規プロセスを悪用する Living Off The Land (LOTL) 攻撃によって検知を回避することを想定しない。LOTL とは例えば Linux において curl コマンドを用いたマルウェアのダウンロード、cron を用いたタスクの永続化、netcat を用いたリバースシェルなどである [10]。

## 5.2 概要

提案システムの概要を図2に示す。本研究ではデータセンタで一台の監視マシンから RDMA を用いて複数台の監視対象マシンのメモリを監視するシステムを提案する。監視マシンは監視対象マシン上で呼び出されたシステムコール呼び出し履歴を収集・マルウェアの検知を行う。

このシステムを構築する上で主に次の二つの技術的課題

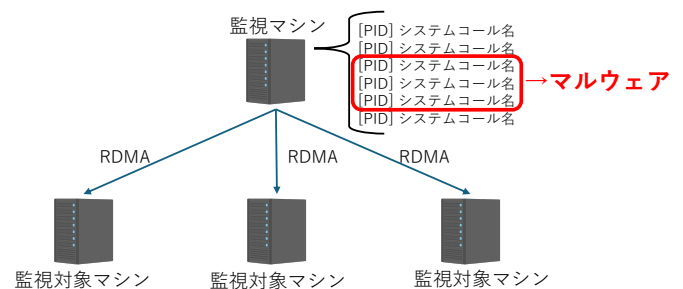


図 2: 提案システムの概要

がある。

- (1) 膨大なシステムコール呼び出し履歴の数の削減
- (2) メモリダンプからのシステムコール呼び出し履歴の復元

本論文では (1) の課題への解決手法としてプロセスベースとシステムコールベースの 2 種類のログの削減手法を提案する。今回 (2) の課題はエージェントプロセスがシステムコール呼び出し履歴を収集することで代用する。システムコール呼び出し履歴などの情報はメモリ上に短時間しか存在せず、RDMA で取得したメモリダンプからの復元は困難である。

## 5.3 設計

データセンタのラック内で1台の監視マシン、その他N台の監視対象マシンが存在し、監視マシンとそれぞれの監視対象マシンはRDMAで通信が可能である。今回システムコール呼び出し履歴はエージェントプロセスが収集することで代用するため、監視対象マシン上でエージェントを実行しシステムコール呼び出し時にエージェントが呼び出し履歴を収集してカーネルが持つバッファに格納する。一定量データがたまったタイミングでカーネルからエージェントにコピーしRDMAで監視マシンへ送信する手法を採用する。監視マシンが監視対象マシンのメモリを読む場合、RDMAで読み込みが完了した通知などの同期処理が必要になり、実装が複雑化するためである。

## 5.4 プロセスベースのフィルタリング手法

プロセススペースのフィルタリングでは多くのシステムコール呼び出しを行う正規プロセスを監視対象から外すことで、送信するデータを削減する。

多くのシステムコール呼び出しを行う正規プロセスを特定するため、システム起動時のプロセスごとのシステムコール呼び出し数の調査を行う。perfで全システムコールのトレースポイントを計測し、イベントデータのバイナリ列からシステムコール名と呼び出し元プロセスのプロセスIDを抽出して1つのログとする。10秒間の計測で得た結果からプロセス毎のログの数を算出し、その結果を元に監視対象から外すプロセスを選択する。

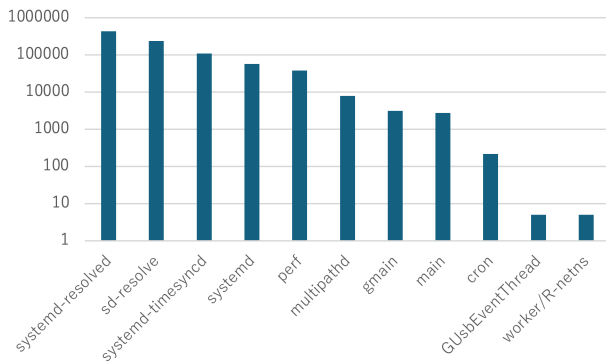


図 3: プロセス別システムコール呼び出し数

調査結果を図3に示す。最もシステムコール呼び出しの多かったプロセスはsystemd-resolvedで、次いでsd-resolve, systemd-timesyncd, systemdの順であった。これらのプロセスはLinuxの起動処理やシステムの管理を行うsystemdが管理しているプロセスである。これらのプロセスが常駐して、多くのシステムコール呼び出しを行っていることが調査の結果から分かった。また、これらのプロセスに次いでシステムコール呼び出しが多かったのはイベントデータを収集するエージェントとして動作するperfであった。

本研究では脅威モデルとしてシステムの正規ツールを悪用する攻撃は想定しないため、systemdに付随するプロセスが呼び出すシステムコールのログは監視対象から外す。perfのプロセスも監視対象から外す。今回システムコール呼び出し履歴を収集する処理をエージェントで代行している。しかし、本研究で想定する環境では監視対象マシンでエージェントは実行されていない。よって以下のプロセスを監視対象外とする。

- (1) systemd
- (2) systemd-resolved
- (3) systemd-timesyncd
- (4) sd-resolve
- (5) systemd-networkd
- (6) systemd-udev
- (7) systemd-logind
- (8) systemd-journald
- (9) perf

### 5.5 システムコールベースのフィルタリング手法

本手法ではマルウェアによって悪用される可能性が高いシステムコールのみをトレースすることで送信すべきデータ量を削減する。

典型的なマルウェアは以下のような挙動で構成される[11]。

- ファイル操作
- 仮想メモリ/ファイルマッピング操作
- ネットワーク操作

表 1: トレースするシステムコール

挙動	システムコール名
ファイル操作	open, close, read, write, ioctl, ftruncate, unlink, openat, unlinkat
仮想メモリ/ファイルマッピング操作	mmap, munmap, mprotect, msync, ptrace, memfd_create
ネットワーク操作	access, socket, connect, accept, bind, listen
プロセス・スレッドの作成/終了	fork, clone, execve, exit
モジュールのロード/アンロード	init_module, delete_module

- プロセス・スレッドの作成/終了
- モジュールのロード/アンロード

これらの挙動を行うシステムコールのログのみを収集する。マルウェアによる挙動とシステムコールの対応表を表1に示す。

## 6. 実装

### 6.1 ログ収集システムの実装

ログの収集はトレースポイントを使用して行う。Linuxでは多くのシステムコールの開始時点にイベント計測やフック処理を行うためのトレースポイントがあらかじめ定義されている。これらのトレースポイントイベントのハンドラとしてログデータを収集してカーネルが持つバッファに書き込む処理を登録する。ユーザ空間のプロセスがバッファからデータを読み込んでRDMA writeでリモートマシンにシステムコールのログが含まれるバイナリデータを送信する。

#### 6.1.1 perf recordの動作

計測対象のイベント、プロセス、CPUを指定してperf recordを実行すると、イベント発生時にイベントを発生させたプロセスのプロセスID、プロセスを実行しているCPU、イベントが発生した時間、命令ポインタなどの情報と共にイベント固有のサンプルデータが格納されているイベントデータがカーネルが管理するリングバッファに書き込まれる。ユーザ空間のperfプロセスはリングバッファからそのデータを読み出し、perf.dataという名前のバイナリファイルに書き込む。perf.dataにはイベントデータの他にメタデータやヘッダ情報も書き込まれる。

#### 6.1.2 perf recordの改造によるエージェントの実装

実装したシステムを図4に示す。今回はトレースポイントイベントのログを収集するエージェントとしてperfを使用する。perf recordがイベント計測結果をperf.dataへ書き込む処理の前にRDMA writeでログを送信する処理を追加する。このときperf.dataに書き込まれるイベントデータ以外のメタデータはマルウェア検知には不要であるため、バッファ内のイベントデータのみを抽出してリモートマシンに送信する。

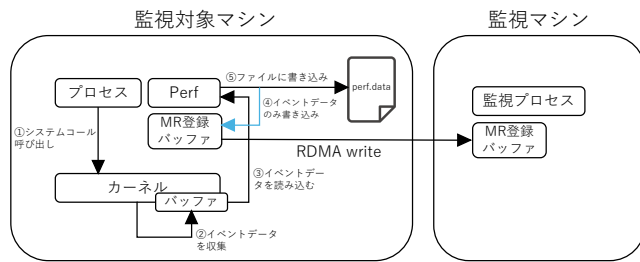


図 4: 実装したシステムの概要

## 6.2 フィルタリング手法の実装

### 6.2.1 プロセスベースのフィルタリング手法の実装

プロセスベースのフィルタリングでは監視対象外のプロセスからのイベントデータをMR登録済みバッファにコピーする際にプロセスIDをもとに判別して削除する。perf recordでシステムコールのトレースポイントイベントを計測する場合、イベントデータのバイナリ列の先頭から64バイト目に呼び出し元プロセスのプロセスIDを表す4バイト分のデータが含まれる[12][13][14][15]。その値と監視対象外のプロセスのプロセスIDを比較して一致すればそのイベントデータをMR登録済みバッファに書き込まないように設定する。プロセスIDをもとに監視対象外のプロセスのイベントデータを削除する処理の概要を図5に示す。この例ではプロセスIDが2,5,11のプロセスを監視対象から外す場合に、それ以外のプロセスのイベントデータのみをMR登録済みバッファに書き込む例を表している。

監視対象外のプロセスのプロセスIDはperf起動時に、/proc上に存在するプロセスごとのディレクトリを順番に探索し、/proc/PID/commに書かれているプロセス名と監視対象外のプロセス名が一致する場合、そのディレクトリ名PIDを配列に追加する。また、sd-resolveは独立したプロセスではなく、systemd-timesyncdから派生したスレッドとして実行される。よって/procディレクトリ内にsd-resolveに対応するプロセスIDのディレクトリが存在しない。そこで本提案システムでは、/proc/PID/-commに格納されている文字列が“systemd-timesyn”の場合のみ、/proc/PID/taskまで参照する。これによってsystemd-timesyncdに属する各スレッドのスレッドIDを列挙し、sd-resolveのスレッドIDを特定する。この配列を入力として、イベントデータが格納されたバイナリから監視対象外のプロセスからのイベントデータを削除する処理を行う。

### 6.2.2 システムコールベースのフィルタリング手法の実装

perfでは収集するイベントを起動時に-eオプションで指定できる。perfでは起動時にperf\_event\_openシステムコールを指定されたイベントのIDを引数として呼ぶことによって、そのイベントが有効化され、そのイベントに対応したファイルのファイルディスクリプタを得る[16]。カーネルはそのファイルを使用してイベントの計測を行

監視対象外のPID 

2	5	11
---	---	----

イベントデータ															
0900	0000	0200	7000	4323	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
e4c8	71d2	5d74	0000	59a2	0000	59a2	0000	0000	0000	0000	0000	0000	0000	0000	0000
5094	9f59	3c64	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0100	0000	0000	0000	0000	3400	0000	fb02	0001	0000	0000	0000	0000	0000	0000	0000
0100	0000	0101	0000	0000	0000	0000	9cff	ffff	0000	0000	0000	0000	0000	0000	0000
0000	0000	931b	a778	9f61	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0900	0000	0200	6000	6223	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
bb73	71d2	5d74	0000	59a2	0000	59a2	0000	0000	0000	0000	0000	0000	0000	0000	0000
05aa	9f59	3c64	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0100	0000	0000	0000	2400	0000	4103	0001	0000	0000	0000	0000	0000	0000	0000	0000
0200	0000	0500	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	e0e1	070c	fd7f	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0900	0000	0200	6000	7223	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
37d9	6ed2	5d74	0000	59a2	0000	59a2	0000	0000	0000	0000	0000	0000	0000	0000	0000
eeb5	9f59	3c64	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0100	0000	0000	0000	2c00	0000	7103	0001	0000	0000	0000	0000	0000	0000	0000	0000
0300	0000	d900	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	00fd	3a84	9f61	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0900	0000	0200	7000	0000	0000	0000	0000	0000	0000	0000	0000

図 5: プロセスのフィルタリング方法

う。本提案手法ではperf recordの実行時にトレース対象とするシステムコールのトレースポイントイベントを指定することで、それらのイベントデータのみを収集する。例としてopen, read, writeシステムコールのみをトレースする場合は以下のようにperf recordを実行することで、各イベントごとにperf\_event\_openシステムコールが呼ばれ、各トレースポイントが有効化される。perf recordでは複数のイベントでリングバッファを共有するため、一回目のperf\_event\_openの戻り値であるファイルディスクリプタを引数にmmapシステムコールを呼び出すことでリングバッファを生成し、そのバッファに残り二つのイベントの計測結果も書き込む。

```
perf record -a -e 'syscalls:sys_enter_open'
-e 'syscalls:sys_enter_read' -e 'syscalls:sys_enter_write'
```

## 6.3 監視プロセスの実装

監視プロセスはRDMA writeによってリモートの監視対象マシンから書き込まれたイベントデータのバイナリ列から呼び出し元プロセスのプロセスIDとシステムコール番号を取り出す。イベントデータの先頭から64バイト目に4バイト分の呼び出し元プロセスのプロセスIDが、68バイト目に4バイト分のシステムコール番号が格納されているため[12][13][14][15]、それらのデータをMR登録済みバッファから取り出す。

システムコール番号とシステムコール名の対応表は監視プロセス起動時に作成する。対応表はシステムコール名とシステムコール番号のマッピングを出力するausyscallコマンドを使用して作成する[17]。ausyscallコマンドを“-dump”というオプションをつけて実行すると全システムコールの番号と名前の対応が出力される。この出力結果からシステムコール番号とシステムコール番号の対応表を作成し、その対応表を参照することでイベントデータから取り出したシステムコール番号に対応するシステムコール名を特定する。

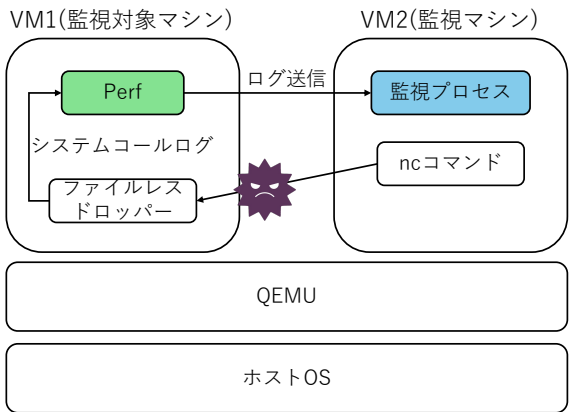


図 6: 実験の構成

## 7. 評価

### 7.1 評価環境

実験を行う環境を表 2 に示す。本実験では QEMU を用いて監視マシンと監視対象マシンとしての VM を 2 台起動し、それらマシンの間で通信を行う。また通信は RDMA をソフトウェア実装である Soft-iWARP を用いて行う。

表 2: 計算機環境

	監視対象マシン	監視マシン
OS	Ubntnu server 24.04.3 LTS	Ubuntu 24.04.2 LTS
カーネル	Linux kernel 6.8.0-88	Linux kernel 6.14.0-37
メモリ	4GB	4GB

### 7.2 実験目的

本実験の目的はシステムコールログのフィルタリングによって、ログの数がどれだけ変化するかを明らかにするとともに、その結果として監視マシンが検知できるマシンの台数を増やすことが可能かどうかを評価することである。

### 7.3 評価方法

評価方法の概要を図 6 に示す。まず監視マシンでログを収集する監視プロセスを実行し、監視対象のマシンでエージェントプロセスである perf コマンドを 30 秒間実行する。またあらかじめペイロードを配信するサーバーとして nc コマンドを任意のポートで待ち合わせておくように実行する。この状態で監視対象のマシンでマルウェアを実行し、監視マシンに送信されたログの数を計測する。これらの評価手順を以下に示す。

- (1) 監視マシンで監視プロセスを実行、以下のコマンドでペイロードを配信するプロセスを実行  
`nc -l 8888 < malware_test`
- (2) 監視対象マシンでエージェントを 30 秒間実行
- (3) 監視対象マシン上でエージェント実行中にマルウェア

表 3: 実験結果

	全ログ数	マルウェアからのログ数	マルウェアからのログの割合
ケース 1	7637122	246346	3.23%
ケース 2	962413	344113	35.8%
ケース 3	870255	416167	47.8%
ケース 4	28116236	460742	1.64%

を実行

今回はマルウェアとして memfd\_create を使ったファイルレスマルウェアのドロップパープロセスを使用する [18]。

また本実験では以下の 4 つのケースで評価を行う。

- ケース 1.** フィルタリングなしで他のプロセスを実行せずにマルウェアのみを実行
- ケース 2.** フィルタリングありで他のプロセスを実行せずにマルウェアのみを実行
- ケース 3.** フィルタリングありでマルウェア “systemd” という名前でコンパイルして実行
- ケース 4.** フィルタリングありでマルウェアの他にも多くのシステムコールを呼び出す dd コマンドを以下のよう

```
dd if=/dev/zero of=/dev/null bs=1
count=500k
```

### 7.4 結果

評価結果を表 3 に、各ケースでのプロセスごとのログ数を図 7 に示す。

ケース 1 では systemd 関連プロセスから大量のシステムコールが呼び出され、監視マシンが受信したログの数は 7637122 件であった。全ログに含まれるマルウェアからのログの割合は 3.23% であった。

次にケース 2 ではフィルタリングによって systemd 関連プロセスからのログがなくなり、監視マシンが受信したログの数は 962413 件であった。全ログに含まれるマルウェアからのログの割合は 35.8% であった。

次にケース 3 ではマルウェアを systemd という名前でコンパイルして実行しているため、systemd という名前のプロセスがマルウェアである。監視マシンが受信したログの数は 870255 件であった。全ログに含まれるマルウェアからのログの割合は 47.8% であった。

次にケース 4 では dd コマンドによるログの数が全プロセスの中で最も多くなり、監視マシンが受信したログの数は 28116236 件であった。全ログに含まれるマルウェアからのログの割合は 1.64% であった。

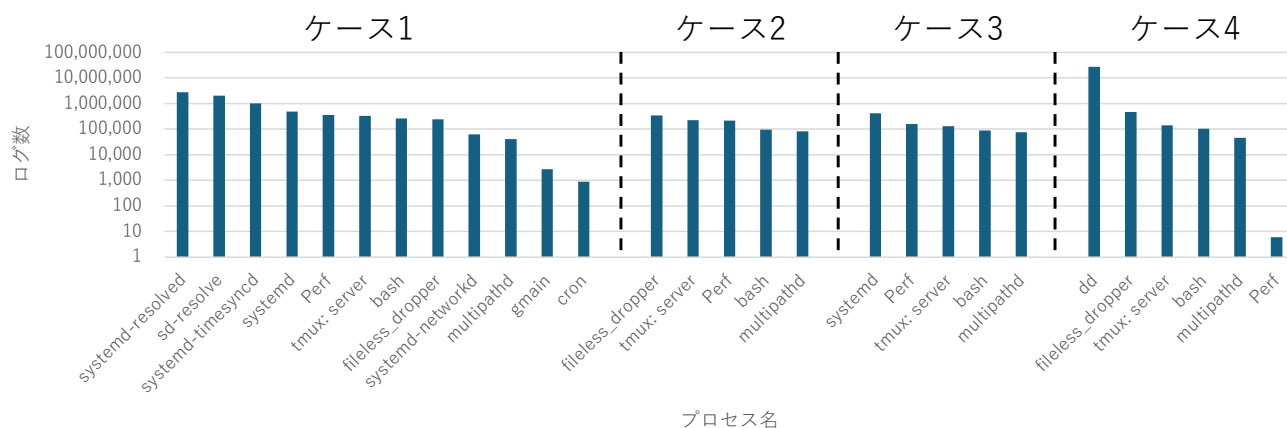


図 7: プロセスごとのログ数

## 8. 考察・今後の課題

### 8.1 考察

#### 8.1.1 フィルタリングの効果

ケース 1 とケース 2 の結果を比較すると、監視マシンが受信したログの総数は約 7.9 倍削減された。またケース 2 では全プロセスの中で最もログの数が多かったプロセスがマルウェアだった。この結果から systemd 関連プロセスからのログがフィルタリング前のログ全体の大部分を占めており、これを消すことでマルウェア以外の多くのプロセスからのログを削除できたと考えられる。これによって監視プロセスがより多くのマシンからのログを収集し、マルウェアを検知できるようになると考えられる。

#### 8.1.2 検知回避の可能性

ケース 3 ではマルウェアを systemd という監視対象外のプロセスの名前でコンパイルすることで、フィルタリングによってマルウェアからのログが収集されないという結果を想定していたが、systemd という名前のマルウェアからのログが確認され、マルウェアからのログの割合もケース 2 と同程度の結果になった。これは監視対象外のプロセスのプロセス ID をエージェント起動時に一回だけ取得していることが要因である。perf がイベントを計測している間に実行されたマルウェアはたとえ名前が監視対象外の systemd であっても、そのプロセスのプロセス ID は監視対象外のプロセス ID のリストには含まれていないため、ログが取得できる。よってマルウェアを監視対象外の正規プロセスの名前で実行した場合でも、マルウェアはリモートマシンの監視プロセスによって検知可能であると考えられる。

ケース 4 では監視マシンが受信したログの総数は 28116236 件となり、ケース 2 の結果と比較して大幅にログ数が増加した。これは多くの read, write システムコールを呼び出す dd コマンドを実行したことが原因だと考え

られる。よって攻撃者によって dd コマンドなどの多くのシステムコール呼び出しを行うプロセスが実行された場合、ログの数が増加し監視プロセスが監視できるマシンの台数が減少してしまう可能性があると考えられる。

### 8.2 今後の課題

提案手法の設計によって監視対象外プロセスのプロセス ID を取得した後に、systemd 関連のプロセスや perf から子プロセスが呼ばれた場合、そのログをフィルタリングによって削除できない。よってケース 2, 3, 4 ともにフィルタリング後も perf からのログが残っているのは perf が計測を行なっている 30 秒間の間に perf によって子プロセスが生成されたことが原因であると考えられる。

また攻撃者によって dd コマンドなどの多くのシステムコール呼び出しを行うプロセスを呼び出された場合、ログの数が増加しリモートマシン上の監視プロセスの負荷が増加するが、dd コマンドは短時間の間に同一の処理を行うシステムコールを高頻度で呼び出すプロセスであるため、イベントデータに含まれる実行された時間、CPU、引数などの情報を元にログをまとめて送信することが可能であると考えられる。これによって送信するログの数を削減し、リモートマシンにかかる負荷を軽減できると考えられる。

また正規ツールを悪用する LOTL 攻撃が行われた場合、このシステムで検知するには困難である。5.1 章で述べたように Linux を標的とした LOTL 攻撃では cron などのツールを使ってプロセスの永続化などを図る。今回作成したシステムでは cron を監視対象から外していないが、systemd でもプロセスの永続化が可能である。systemd を悪用されて攻撃された場合、この提案システムでは検知できない。このような攻撃に対処するためにより適切なログ削減手法を模索する必要がある。

## 9. 結論

データセンタ内をラテラルムーブメントするマルウェアを単一ホスト上のエージェントで検知することは困難である。マルウェアはエージェントを改竄・停止することにより検知を回避する可能性がある。本研究では RDMA を用いたメモリ監視によるマルウェア検知システムを提案し、監視マシンが受信するデータ量削減のためにプロセスベースとシステムコールベースの2種類のデータ量削減手法を提案した。提案手法によって監視マシンが受信するデータ量を約 7.9 倍削減させることに成功した。

## 参考文献

- [1] トレンドマイクロ：アサヒグループへのランサムウェア攻撃事例を記者会見から考察～今、注意すべき「データセンター」への侵害, [https://www.trendmicro.com/ja\\_jp/jp-security/25/1/expertview-20251218-01.html](https://www.trendmicro.com/ja_jp/jp-security/25/1/expertview-20251218-01.html) (2025).
- [2] トレンドマイクロ：ランサムウェア「Kasseika」による BY-OVD 攻撃: リモート管理ツール「PsExec」やドライバ「Martini」を不正利用, [https://www.trendmicro.com/ja\\_jp/research/24/b/kasseika-ransomware-deploys-byovd-attacks-abuses-psexec-and-expl.html](https://www.trendmicro.com/ja_jp/research/24/b/kasseika-ransomware-deploys-byovd-attacks-abuses-psexec-and-expl.html) (2024).
- [3] SOMPO CYBER SECURITY: ラテラルムーブメントとは【用語集詳細】, <https://www.sompocybersecurity.com/column/glossary/lateral-movement> (2018).
- [4] Fang, Y., Wang, C., Fang, Z. and Huang, C.: LM-Tracker: Lateral movement path detection based on heterogeneous graph embedding, *Neurocomputing* (2022).
- [5] 吉川孝志: マルウェアの教科書, 日経 BP (2023).
- [6] Nair, S. J. and Syam, S. R.: Automated Malware Detection Using Memory Forensics, *15th International Conference on Computing Communication and Networking Technologies (ICCCNT)* (2024).
- [7] Oliveri, A. and Balzarotti, D.: A Comprehensive Quantification of Inconsistencies in Memory Dumps (2025).
- [8] Liu, H., Xing, J., Huang, Y., Zhuo, D., Devadas, S. and Chen, A.: Remote Direct Memory Introspection, *32nd USENIX Security Symposium*, pp. 6043–6060 (2023).
- [9] Gaber, M., Ahmed, M. and Janicke, H.: Defeating evasive malware with Peekaboo: Extracting authentic malware behavior with dynamic binary instrumentation, *Journal of Information Security and Applications* (2025).
- [10] GTF0Bin: GTF0Bin, <https://gtfobins.org/>.
- [11] 大月 勇人, 瀧本 栄二, 檜山 武浩, 毛利 公一: マルウェア挙動解析のためのシステムコール実行結果取得法, *Computer Security Symposium 2011*, pp. 95–100 (2011).
- [12] GitHub: linux/include/uapi/linux/perf\_event.h at master · torvalds/linux, [https://github.com/torvalds/linux/blob/master/include/uapi/linux/perf\\_event.h#L840](https://github.com/torvalds/linux/blob/master/include/uapi/linux/perf_event.h#L840).
- [13] GitHub: linux/kernel/events/core.c at master · torvalds/linux, <https://github.com/torvalds/linux/blob/master/kernel/events/core.c#L7889>.
- [14] GitHub: linux/include/linux/perf\_event.h at master · torvalds/linux, [https://github.com/torvalds/linux/blob/master/include/linux/perf\\_event.h#L1293](https://github.com/torvalds/linux/blob/master/include/linux/perf_event.h#L1293).
- [15] The Linux Kernel Archives: Event Tracing, <https://www.kernel.org/doc/html/latest/trace/events.html>.
- [16] man7.org: perf\_event\_open(2) - Linux manual page, [https://man7.org/linux/man-pages/man2/perf\\_event\\_open.2.html](https://man7.org/linux/man-pages/man2/perf_event_open.2.html).
- [17] Die.net: ausyscall(8) - Linux man page, <https://linux.die.net/man/8/ausyscall>.
- [18] 0x00pico: 00sec Droppers, <https://archive.0x00sec.org/t/super-stealthy-droppers/3715> (2017).