システムコールの監視を利用した メモリフォレンジックによる鍵回復手法の提案

樋口 諒[1,a)]　稲村 浩[2]　石田 繁巳[2]

概要：近年，ランサムウェアによる被害が増加しており，その対策が必須となっている．警察庁によると，バックアップの取得によりランサムウェア暗号化攻撃から被害直前の水準まで復元できたケースは 3 割未満である．バックアップを利用しない手法としてメモリフォレンジックによる鍵の回復手法が知られている．この手法は，ランサムウェアが使用しているメモリから暗号鍵を取得し被害ファイルを復元する．メモリフォレンジックではメモリ上に暗号鍵が存在するタイミングでメモリを調べることが重要である．本稿では，鍵の復元可能性の向上を目的とし，囮ファイルに対する `write` システムコールを監視することで，ランサムウェアプロセスを検知・特定し，ランサムウェアプロセスを停止させメモリダンプする手法を提案する．囮ファイル 1 つに対して暗号化攻撃を行い，ランサムウェアプロセス停止の有無によって復元の可否を確認することで提案手法を評価した．その結果，プロセスを一時停止させることでメモリ内容の一貫性を確保した状態でのメモリダンプが可能となり，暗号鍵を回復することが確認できた．しかしながら，囮ファイルへの `write` システムコールの監視をすり抜けるケースも見られた．

1. はじめに

近年，ランサムウェアによる被害が増加しており，脅威となっている．警察庁サイバー警察局によると，令和 6 年における企業や団体に対するランサムウェアによる暗号化攻撃は 222 件発生しており，高水準で推移している [1]．

ランサムウェア被害拡大の背景には，ランサムウェアの開発・運営を行う者が攻撃実行者にランサムウェアを提供し，見返りとして金銭を要求する RaaS（Ransomware as a Service）による影響が強く出てきている．これにより，専門知識を持たない者でもランサムウェアを手にしやすく，攻撃を行いやすくなった．

ランサムウェア被害の対策として，機械学習によるランサムウェアの早期検知手法が存在する [2]．しかし，ランサムウェアを早期検知し暗号化攻撃を停止しても，暗号化されたファイルは存在した．これは，暗号化攻撃の高速化などが要因として挙げられる [3]．そのため，ランサムウェアを検知する手法と合わせて，被害から復元する手法も必要となってくる．

ランサムウェア被害などのインシデントからの回復として多くの組織や企業でバックアップを取得している．実際，警察庁サイバー警察局によると，ランサムウェア被害にあった組織や企業のうち約 9 割がバックアップを取得していた．しかし，ランサムウェア暗号化攻撃から被害直前の水準まで復元できた組織や企業は 3 割にも満たない．[1]．これは，バックアップの運用不備やバックアップそのものの暗号化などが原因として挙げられている．このことからバックアップは過信できないため，バックアップを利用しない復元手法も重要であると言える．

バックアップを利用しない復元手法として，メモリフォレンジックによる鍵回復手法が存在する．鍵回復手法とは，ランサムウェアなどが暗号化に用いた暗号鍵を何らかの方法で取得することで，暗号化されたファイルなどを復元する手法である [4]．鍵回復手法には様々な手法が存在するが，その 1 つに，ランサムウェアが利用するメモリ内容を解析することで鍵回復を行う「メモリフォレンジック」が存在する．メモリフォレンジックによる鍵回復手法では，ランサムウェアが動作している間にメモリダンプし，取得したメモリイメージを解析することで暗号鍵を取得して復元する．メモリフォレンジックによる鍵回復手法ではメモリダンプするタイミングが重要となってくる．なぜならば，メモリ上に暗号鍵が存在しないタイミングでメモリダンプし解析しても，意味がないためである．

本稿では，メモリフォレンジックによる鍵回復手法の復元可能性の向上を目的とする．そのためには，ランサムウェア暗号化攻撃の最中にメモリダンプする必要がある．

[1] 公立はこだて未来大学大学院システム情報科学研究科
[2] 公立はこだて未来大学システム情報科学部
[a)] g2125068@fun.ac.jp

© 2025 Information Processing Society of Japan

ランサムウェア暗号化攻撃の検知に，囮ファイルと呼ばれる正規のユーザが操作しない前提を置いたファイルを利用する．囮ファイルへのアクセスを監視することで，囮ファイルが操作された際にランサムウェア暗号化攻撃を検知することができる．評価では，擬似ランサムウェアによって暗号化攻撃を行い，囮ファイル監視の成功と復元の可否により評価する．

本稿の構成は以下の通りである．第 2 章では，本研究に関わる関連研究について紹介する．第 3 章では，提案手法について述べた後，第 4 章で評価実験について述べる．最後に，第 5 章でまとめとする．

2. 関連研究

本章では，本研究に関連する，「メモリフォレンジックによる鍵回復の研究」，「囮ファイルによるランサムウェア暗号化攻撃の防御に関する研究」，「囮ファイルを利用したメモリフォレンジックによる鍵回復の研究」の 3 つの研究について紹介する．

2.1 メモリフォレンジックによる鍵回復の研究

Davies らは，ランサムウェア検体の解析により，ランサムウェアの内部処理によるメモリの変化の時系列であるタイムラインを作成することでメモリダンプのタイミングを推定する手法を提案した [5]．結果として，取得した暗号鍵により攻撃を受けたファイルを復元できていた．タイムラインの作成は，実際にランサムウェア検体を動作させ，メモリの状態を監視することで作成していた．

この手法の課題として，未知のランサムウェアによる暗号化攻撃に対応できないことが挙げられる．解析とタイムラインの作成が行われていない未知のランサムウェアに対しては，暗号鍵がメモリ上に保持されているタイミングが不明となるため，暗号鍵がメモリ上に保持されているタイミングがわからずに，メモリ上に暗号鍵を保持しているタイミングとメモリダンプするタイミングが重ならず，暗号鍵を含んだメモリダンプを取得できない可能性がある．

2.2 囮ファイルによるランサムウェア暗号化攻撃の防御に関する研究

田中らは，囮ファイルを利用したランサムウェア暗号化攻撃を検知し遅延する手法を提案した [6]．この研究では，囮ファイルを設置し監視することでランサムウェア暗号化攻撃を検知し，暗号化攻撃が行われている間に大量の囮ファイルを作成し設置することで，暗号化攻撃を遅延させた．結果として，多くのランサムウェア検体に対して暗号化攻撃の遅延することができていた．これにより，利用者は攻撃に気付くことでランサムウェアを停止させるなど対処の機会が得られる．

この手法の課題として，囮ファイル暗号化以前に暗号化されてしまった被害ファイルを復元できない点である．この手法では，ランサムウェア検体を解析することで，ランサムウェアが暗号化を開始するディレクトリを割り出し囮ファイルの設置を行なっている．調査した結果として，ランサムウェアが暗号化を開始するディレクトリはランサムウェアごとに異なっていた．そのため，囮ファイルが設置されていないディレクトリから暗号化攻撃されてしまうと，囮ファイルへのアクセスによってランサムウェアを検知した時点で既に暗号化された被害ファイルを復元する手段はない．

2.3 メモリ領域局所化と CPU 使用率を制限する手法

著者らは，バックアップを用いない暗号化攻撃からの復元を目的に研究を進めてきた．確実な回復のために暗号鍵を含むと思われるメモリダンプの取得回数を増やすことを課題として，囮ファイルの監視によりランサムウェアプロセスを検知・特定し，メモリ領域局所化とランサムウェアプロセスの CPU 使用率を制限する手法を提案した [7]．

この手法では囮ファイルを監視することにより，ランサムウェアプロセスの Process ID (PID) を特定する．PID を特定し Linux に存在する`/proc` ディレクトリからメモリイメージを取得することで，ランサムウェアプロセスが使用しているメモリ領域のみをダンプした．`/proc` ディレクトリ下の`/pid/maps` からは，プロセスの仮想メモリ空間のアドレスが取得できるため，これを用いて`/proc/pid/mem` にアクセスし，取得した仮想メモリ空間のアドレスが示す領域のみを取得することで，メモリ領域局所化によるメモリダンプを実装した．これにより，全メモリダンプする既存のツール[*1] と比較して，メモリダンプする領域が縮小したため，単位時間あたりのメモリダンプ可能な回数を増加させた．

一貫性のあるメモリダンプ取得のため，ランサムウェアの動作によるメモリの変更を緩慢にすることを狙って，ランサムウェアプロセスの CPU 使用率を制限した．囮ファイルの監視により PID を特定することでランサムウェアプロセスの CPU 使用率を制限することが可能になった．`cgroup-v2` により予め作成されたグループの中に，ランサムウェアプロセスの PID を追加することで CPU 使用率を制限し，ランサムウェア暗号化処理時間が増加することで，メモリダンプ回数を増加させていた．

この手法には 2 つの課題が存在する．1 つ目は，囮ファイルの監視によるランサムウェアプロセスの検知・特定に関する実装について詳細な設計がなく論文では言及していない点である．特に，この提案手法ではランサムウェアプロセスの PID の取得が重要となってくるが，具体的な実装について明らかになっていない．

[*1] 504ensicsLabs: LiME: Linux Memory Extractor, `https://github.com/504ensicsLabs/LiME`, Accessed: 2025-01-20.

© 2025 Information Processing Society of Japan

2つ目は，ランサムウェアプロセスを停止させずに並行してメモリダンプしている点である．一般的にマルウェアでは，メモリフォレンジックによる解析に対策する手法が知られている [8]．これはランサムウェアにも言えることであり，メモリ上に暗号鍵が読み取り可能な状態ではなるべく存在しないようにしていることが，Davies らの研究で明らかになっている [5]．そのため，ランサムウェアプロセスを停止せずにメモリダンプすると，メモリの状態が変化してしまい，仮に多数の被害ファイルの中で囮ファイルの暗号化が処理の最後に行われた場合には，検知からメモリダンプまでの経過時間の間でランサムウェアが暗号鍵をメモリ上から削除してしまうことが考えられる．この場合にはメモリダンプしても鍵を発見できない．

3. 提案手法

本稿では，囮ファイルを監視することによるランサムウェアプロセスの検知・特定と，ランサムウェアプロセスの停止によるメモリ情報の一貫性を保ったメモリダンプ手法を提案する．提案手法の全体の流れを**図 1** に示す．提案手法では，まず，囮ファイルを監視する．正規のユーザやプロセスがアクセスすることはないファイルであるため，ここにアクセスがあれば異常と見なせる．囮ファイルへのアクセスからランサムウェアによる囮ファイルに対する暗号化攻撃を検知し，ランサムウェアプロセスを特定する．その後に，ランサムウェアプロセスを停止させ，停止している間にメモリダンプを行い，取得したメモリイメージから暗号鍵を取得する．暗号鍵の正誤判定には，囮ファイルの平文が既知であることを利用する．鍵候補によって復元した結果が囮ファイルの平文と一致しているかどうかで正誤判定する．

本章では，図 1 の「①囮ファイルの監視」,「③検知・特定」,「④ランサムウェアプロセスの停止」に関する具体的な実装について述べる．

3.1 囮ファイルの監視による検知・特定

本節では，図 1 の「①囮ファイルの監視」と「③検知・特定」についての具体的な実装について述べる．

図 1 の「①囮ファイルの監視」から「④ランサムウェアプロセスの停止」までの具体的な流れを**図 2** に示す．アルゴリズム 1 は，「①囮ファイルの監視」から「④ランサムウェアプロセスの停止」までの実装を示している．「①囮ファイルの監視」は，1. カーネル空間側で全ての `write` システムコールを監視，2. システムコールを呼び出したプロセスの PID をユーザ空間に送信，3. PID が持つファイルディスクリプタの中に囮ファイルが存在するか確認の 3 ステップによって行う．

囮ファイルへのアクセスの監視について `write` システムコールの発行をトレースすることで，囮ファイルへ書き込みをしたランサムウェアプロセスの検知・特定を行う．プロセスがファイルに対して実際のデータを書き込む際に，`write` システムコールは発生する．このため，`write` システムコールの監視によりファイル内容の変更が発生する直前のタイミングを正確に捉えることが可能である．また，`read` システムコールなどは悪意のあるプロセスかどうかの判定が難しい．以上の点から，`write` システムコールの監視によってランサムウェアプロセスの検知・特定を実現する．

`write` システムコールの監視には extended Berkeley Packet Filter (eBPF) を利用する．eBPF は，ネットワークのパケットフィルタリングを目的に利用されていた BPF を拡張し，Linux カーネル内での様々なイベントを制御できるようにしたものである．eBPF では，カーネル空間内で特定ファイルへのアクセスのみをフィルタリングする機能がそのままでは利用できない．そこで本稿では，全ての `write` システムコールエントリーの監視をカーネル空間側で行い，`write` システムコールエントリーを行ったプロセスの PID 情報をユーザ空間に送信し，ユーザ空間側において送信された PID が囮ファイルのファイルディスクリプタを保持しているか判別することで，囮ファイルに対する `write` システムコールエントリーを監視する．

カーネル空間とユーザ空間の実装には，BPF Compiler Collection (BCC) を利用する [9]．BCC とは，効率的なカーネルトレースおよび操作プログラムを作成するためのツールキットであり，eBPF を利用している．BCC を利用することにより，ユーザ空間の Python スクリプト内に eBPF コードを埋め込むことができ，Python プログラムの実行時に eBPF コードがコンパイルされカーネル空間内で実行される．また，カーネル空間とユーザ空間からアクセスできる共有データ構造の定義などを行うことができる．これにより，カーネル空間側で全ての `write` システムコールをフックし，`write` システムコールを要求したプロセスの PID をユーザ空間側に送信する．

Python により実装したユーザ空間側では，カーネル空間から送られてきた PID を元に，プロセスが保持しているファイルディスクリプタ情報を確認する．PID からプロセスが持つファイルディスクリプタの確認には，Linux に存在する`/proc` ディレクトリ下に存在する`/proc/pid/fd` を利用する．Linux に存在する`/proc` ディレクトリには，プロセスに関する情報が PID により管理されており，その中に存在する`/proc/pid/fd` には，プロセスが保持しているファイルディスクリプタに関するシンボリックリンクが保持されている．これを利用することで，プロセスの PID を取得により，プロセスが保持しているファイルディスクリプタのシンボリックリンクを獲得し，囮ファイルのファイルディスクリプタを保持しているのかを判定できる．もしも，囮ファイルのァイルディスクリプタを保持している

© 2025 Information Processing Society of Japan

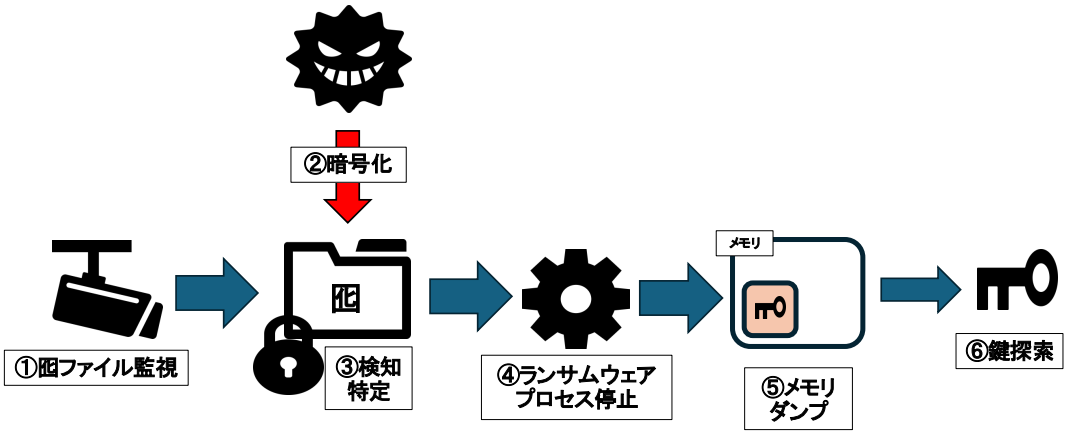

図 1: 提案手法の流れ

場合，そのプロセスがランサムウェアプロセスであると検知できる．この時，既に PID は囮ファイルにアクセスしているのか確認するために取得しているため，既存研究で提案されたメモリ領域局所化したメモリダンプや，プロセス停止を行うことができる．

3.2 ランサムウェアプロセス停止のメモリダンプ

本節では，図 1 の「④ランサムウェアプロセスの停止」についての具体的な実装について述べる．

本稿では，eBPF を利用した囮ファイルの監視によるランサムウェアプロセスの検知・特定により，ランサムウェアプロセスの PID を取得している．そのため，メモリダンプする際にランサムウェアプロセスを停止させ，一貫性のあるメモリ情報を取得することができる．ランサムウェアプロセスの停止には，SIGSTOP シグナルを利用する．シグナルは，プロセスに対して非同期的に送信される通知機構であり，ユーザからの操作によって，プロセスを制御することができる．本稿では，ランサムウェアプロセスの停止に SIGSTOP シグナルを送ることで，一時停止させる．これは，ランサムウェアプロセスを強制的に一時停止させたいので，シグナルブロックができない SIGSTOP を採用する．SIGSTOP シグナルは，`write` システムコールを要求したプロセスが囮ファイルのファイルディスクリプタを保持しているか確認をしているユーザ空間側からランサムウェアプロセスに送信する．

図 1 の「⑤メモリダンプ」と「⑥鍵探索」には，著者らが提案したメモリ領域局所化手法を利用して行う [7]．

4. 評価

本章では，提案したシステムの評価について記述する．

Algorithm 1 囮ファイル書き込み検知時のメモリダンプ

```
1: function MonitorAndDump(TARGET_FILE)
2:     CURRENT_PID ← getpid
3:     Initialize eBPF program with trace_write_entry()
4:     Attach eBPF to __x64_sys_write
5:     while true do
6:         event ← wait_for_perf_event
7:         pid ← event.pid
8:         if pid == CURRENT_PID then
9:             continue
10:        end if
11:        fd_paths ← get_fd_paths(pid)
12:        if TARGET_FILE ∈ fd_paths then
13:            SendSignal(pid, SIGSTOP)
14:            regions ← get_memory_maps(pid)
15:            for all (start, end, type) ∈ regions do
16:                output ← pid_type_.bin
17:                dump_memory(pid, start, end, type, output)
18:            end for
19:            SendSignal(pid, SIGCONT)
20:        end if
21:    end while
22: end function
```

本稿では，囮ファイル監視によるランサムウェアプロセスの検知・特定と，ランサムウェアプロセスの停止によるメモリ情報の一貫性を持ったメモリダンプの提案を行った．提案手法の実現可能性と有効性を確認するために，疑似ランサムウェアによる暗号化攻撃からの被害ファイルの復元可否を，ランサムウェアプロセス停止の有無で比較することにより評価した．被害ファイルの復元にはメモリダンプからの暗号鍵の取得が必要であり復元成功回数の比較により手法の有効性を確認する．

同時に，提案した囮ファイル監視によるランサムウェアプロセスの検知手法についてよる囮ファイルの監視の成否

© 2025 Information Processing Society of Japan

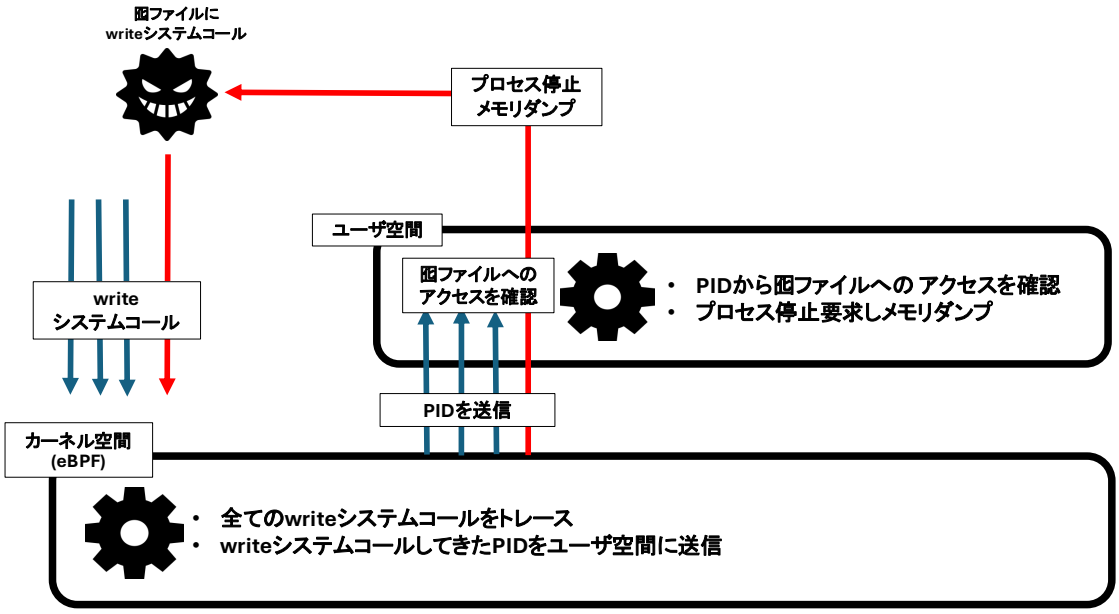

図 2: 囮ファイル監視システムの流れ

も評価した．実装上，監視部が通知された PID から囮ファイルアクセスの有無を確認する前にランサムウェアプロセスがファイル処理を終えてしまう可能性がある．このような監視のすり抜け状況の発生の有無を確認する．

実験では，eBPF を利用した囮ファイルの監視によるランサムウェアプロセスの検知・特定までを共通に行ない，その後のメモリダンプ処理の際に，先行研究 [7] による，停止させずランサムウェアプロセスの CPU 使用率を制限する場合と，提案手法によるランサムウェアプロセス停止の場合とで，復元の可否を確認した．

4.1 実験環境

本稿では，ホスト OS の Windows 11 上に VirtualBox によって Ubuntu 22.04 を仮想マシンとしてインストールし実験を行った．ホスト OS である Windows 11 に VirtualBox を用いて Ubuntu を仮想化することで，テストや開発環境を独立して動作させた．仮想マシンには 4GB のメモリと 1 つのコアを割り当てた．**表 1** にホスト OS 及び仮想マシンの仕様を示す．

表 1: ホスト OS および仮想マシンの仕様

項目	仕様
ホスト OS	Windows 11
ホストメモリ	16GB
ホスト CPU	Intel Core i7
仮想マシン OS	Ubuntu 22.04
仮想メモリ	4GB
仮想 CPU コア数	1

4.2 囮ファイル

囮ファイルは先行研究を参考にし，囮ファイルのファイルサイズを 2KB に設定した [6, 7]．ファイルの内容は乱数で構成した平文を 2KB 分割り当てた．本実験では，攻撃対象としたディレクトリに囮ファイル 1 個を設置し，暗号化攻撃を行った．

4.3 擬似ランサムウェア

実験では，実際のランサムウェアの動作を模した擬似ランサムウェアを利用した．擬似ランサムウェアは，先行研究を参考に Python により実装した [6, 7]．擬似ランサムウェアは以下の手順で暗号化攻撃を行うようにした．

(1) 暗号化対象ファイルを読み取る
(2) 暗号化対象ファイルを暗号化する
(3) 暗号化後データを書き込む

ランサムウェアの暗号化アルゴリズムは AES 方式とし，鍵長は 256 bit とした．擬似ランサムウェアは暗号化終了と同時に暗号鍵をメモリ上から破棄して直ちに終了させた．

4.4 ランサムウェアプロセス停止の有無による鍵回復の評価

囮ファイル 1 つに対して擬似ランサムウェアによって暗号化攻撃を行い，eBPF による囮ファイルの監視によるランサムウェアプロセスの検知・特定下の中で，ランサムウェアプロセス停止する場合と，ランサムウェアプロセスを停止せずに CPU 使用率制限する場合とで，復元の可否，メモリダンプの可否などを比較することで評価する．

具体的には，CPU 使用率上限ごとに復元成功回数，ダン

© 2025 Information Processing Society of Japan

プ成功回数，ダンプ部分成功回数，監視すり抜け回数，ダンプ失敗回数によって示す．ダンプ部分成功回数は，メモリダンプ処理が途中まで成功した回数を示す．監視すり抜け回数は，eBPF による囮ファイルへのアクセス監視が失敗した回数を示す．ダンプ失敗回数は，監視によりランサムウェア暗号化攻撃を検知することはできたが，メモリダンプが間に合わなかった回数を示す．ダンプ成功回数は想定されるサイズでメモリダンプが終了したことを示す．ただしダンプ内容の一貫性が維持されているかどうかはここでは問わない．メモリを走査して得た鍵回復の成功回数で評価する．

ランサムウェアプロセスを停止せず CPU 使用率を制限する場合では，先行研究を参考に CPU 使用率制限を cgroup-v2 により行い，CPU 使用率の上限の設定は先行研究と同様に，100%，80%，60%，40%，20%，10%，1% に設定する [7]．CPU 使用率の制限は，最初に cgroup-v2 に制限するプロセスを格納するディレクトリを作成しておき，提案手法でランサムウェアプロセスを停止するために，ランサムウェアプロセスに対して SIGSTOP シグナルを送信する代わりに，CPU 使用率を制限するランサムウェアプロセスを作成したディレクトリに追加することで行った．

プロセスを停止する場合と CPU 使用率を制限する場合の各設定のそれぞれで 20 回試行した．

4.4.1 プロセスを停止しない場合の結果

表 2 に，CPU 使用率を制限する場合に対して，擬似ランサムウェアにより 1 つの囮ファイルを暗号化攻撃する試行を 20 回行った結果を示す．プロセスを停止せず CPU 使用率の制限によって低速で実行を継続させつつメモリダンプする場合では，後述する eBPF による囮ファイルの監視のすり抜け以外に，メモリダンプ処理が間に合わずに復元できないケースが少数ながら存在した．これは，プロセスを停止しておらず動作し続けていることが原因である．CPU 使用率制限してもプロセスは動作し続けてしまうため，ランサムウェアプロセスが鍵を含む領域のメモリダンプ処理の完了前に実行を終了してしまい，メモリダンプが間に合わなかったためである．また，メモリダンプ処理がランサムウェアプロセス終了前に間に合ったが，メモリイメージ内に暗号鍵が存在せずに復元に失敗するケースも存在した．こちらも同様に，ランサムウェアの検知が遅くなってしまうと，仮にメモリダンプできたとしても，既に暗号鍵をメモリ上から破棄してしまっているために復元できなかった．制限した CPU 使用率による結果の違いがあまりみられなかったのは，実験では 2KB の囮ファイル 1 つのみに対して攻撃を行っているため，CPU 使用率制限による影響が小さかったことが考えられる．

4.4.2 提案手法によるプロセス停止を用いた結果

提案手法にてランサムウェアプロセスを停止させたことによって，メモリダンプが間に合わないケースは発生せず，ファイルの復元にも成功していることからダンプ情報の一貫性も得られたと考えられる．

提案手法に対して擬似ランサムウェアにより 1 つの囮ファイルを暗号化攻撃した結果を**表 3** に示す．プロセスを停止しメモリダンプする提案手法では，ランサムウェア暗号化攻撃の検知が行われた場合ではメモリダンプと復元が全て成功した．プロセスを停止する提案手法では，検知した際に直ちにプロセスを停止させるため，メモリ状態が変化せずにメモリダンプできたことから，途中までの部分成功や失敗が存在しなかった．

今回の実験では，先行研究によるメモリ領域局所化を利用したメモリダンプを利用したが，ランサムウェアがマルチプロセスにより暗号化を行う場合や，より詳細に解析を行う場合において，単一プロセスのメモリ領域をダンプするより全てのメモリ領域をダンプする方が良いことがある．全てのメモリ領域をダンプする際には，CPU 使用率制限により遅延してもダンプする領域が大きくてメモリダンプが間に合わないケースが存在する可能性があるため，今回のようにランサムウェアプロセスを停止しメモリダンプすることが効果的である．

4.5 提案手法による囮ファイル監視を用いた検知の評価

先行研究の手法と提案手法に共通して囮ファイルの監視に失敗しすり抜けるケースが存在した．表 2 と表 3 に示すように，ランサムウェアプロセスを停止する場合とランサムウェアプロセスの CPU 使用率を制限する場合のそれぞれで，囮ファイルの監視ををすり抜ける場合が見られた．これは，図 2 においてユーザ空間側で /proc/pid/fd の情報を確認し囮ファイルかどうか判定する処理にかかる前に囮ファイルのファイルディスクリプタをランサムウェアプロセスが消去していることで発生している．

5. 終わりに

ランサムウェアによる暗号化攻撃に対するバックアップを利用しない復元手法としてメモリフォレンジックによる鍵回復手法に着目した．鍵の復元可能性の向上を目的とし，ランサムウェアプロセスの検知・特定方法と，メモリ情報の一貫性を保ったメモリダンプ方法の実現を課題とした．eBPF による囮ファイルの監視によるランサムウェアプロセスの検知・特定機能の実装方法と，プロセス停止によるメモリダンプ手法を提案した．

先行研究 [7] では，囮ファイル監視によりランサムウェア暗号化攻撃を検知・特定し，CPU 使用率制限によりメモリダンプする手法を示したが，本稿では，メモリフォレンジックによる鍵回復の可能性を向上させるために，プロセスを停止させメモリダンプする手法とした．ランサムウェアが鍵をメモリ上に保持しているタイミングでプロセスを停止し，メモリダンプするために，囮ファイルへの write

© 2025 Information Processing Society of Japan

表 2: CPU 制限ごとのメモリダンプ結果集計

CPU 使用率上限 (%)	復元成功回数 (回)	ダンプ成功回数 (回)	ダンプ部分成功回数 (回)	監視すり抜け回数 (回)	ダンプ失敗回数 (回)
1%	10	12	0	6	2
10%	10	11	0	8	1
20%	10	11	1	8	0
40%	7	9	4	6	1
60%	11	10	3	6	1
80%	9	10	1	6	3
100%	8	9	1	7	3

表 3: プロセス停止によるメモリダンプ結果集計

手法	復元成功回数 (回)	ダンプ成功回数 (回)	ダンプ部分成功回数 (回)	監視すり抜け回数 (回)	ダンプ失敗回数 (回)
プロセス停止	15	15	0	5	0

システムコール発行のタイミングでプロセスを停止する手法を提案した．

提案手法にてランサムウェアプロセスを停止させた場合と，先行研究での完全には停止せず CPU 使用率制限する場合において，囮ファイル 1 つに対して擬似ランサムウェアにより暗号化攻撃し，復元の可否を確認することで評価を行った．同時に，提案手法による囮ファイルの監視が失敗し，ランサムウェアによる暗号化攻撃を見逃がすケースが存在しないかの確認も行った．

提案手法にてランサムウェアプロセスを停止させたことによって，メモリダンプが間に合わないケースは発生せず，ファイルの復元にも成功していることからダンプ情報の一貫性も得られたと考えられる．しかし，先行研究の手法と提案手法に共通して囮ファイルの監視に失敗しすり抜けるケースが存在した．今後は，監視のすり抜けを減らす方法などの検討を進める．

参考文献

[1] 警察庁サイバー警察局. 令和 6 年におけるサイバー空間をめぐる脅威の情勢等について, March 2025. https://www.npa.go.jp/publications/statistics/cybersecurity/data/R6/R06_cyber_jousei.pdf.

[2] Daniel Morato, Eduardo Berrueta, Eduardo Magaña, and Mikel Izal. Ransomware early detection by the analysis of file sharing traffic. *Journal of Network and Computer Applications*, Vol. 124, pp. 14–32, 2018.

[3] Splunk. Gone in 52 seconds and 42 minutes: A comparative analysis of ransomware encryption speed. https://www.splunk.com/en_us/blog/security/gone-in-52-seconds-and-42-minutes-a-comparative-analysis-of-ransomware-encryption-speed.html, 2024.

[4] Pranshu Bajpai, Aditya K Sood, and Richard Enbody. A key-management-based taxonomy for ransomware. In *2018 APWG Symposium on Electronic Crime Research (eCrime)*, pp. 1–12. IEEE, 2018.

[5] Simon R Davies, Richard Macfarlane, and William J Buchanan. Evaluation of live forensic techniques in ransomware attack mitigation. *Forensic Science International: Digital Investigation*, Vol. 33, p. 300979, 2020.

[6] 田中智也, 小池一樹, 小林良太郎, 加藤雅彦ほか. ダミーファイルを利用した暗号化型ランサムウェア対策システムの実装. コンピュータセキュリティシンポジウム 2019 論文集, Vol. 2019, pp. 163–169, 2019.

[7] 樋口諒, 稲村浩, 石田繁巳. ランサムウェア暗号化攻撃遅延とメモリ領域局所化を利用した鍵回復手法の初期検討. 情報処理学会第 87 回全国大会講演論文集, 2025.

[8] Ralph Palutke, Frank Block, Patrick Reichenberger, and Dominik Stripeika. Hiding process memory via anti-forensic techniques. *Forensic Science International: Digital Investigation*, Vol. 33, p. 301012, 2020.

[9] IO Visor Project. BCC: BPF Compiler Collection, 2025. https://github.com/iovisor/bcc.

© 2025 Information Processing Society of Japan