卒業論文　2019 年度（令和元年度）

RDMA を用いたメモリ探索による
遠隔ベアメタルマシンのプロセス情報の取得

慶應義塾大学 環境情報学部

石川 達敬

徳田・村井・楠本・中村・高汐・バンミーター・植原・三次・中澤・武田
合同研究プロジェクト

2020 年 1 月

卒業論文　2019 年度（令和元年度）

RDMA を用いたメモリ探索による
遠隔ベアメタルマシンのプロセス情報の取得

論文要旨

大規模データセンターなど，大量の物理的なコンピュータの管理者は，通常時の監視に加えて緊急時の原因究明など，コンピュータを解析する場面に直面する．しかし，顧客にコンピュータを貸し出している場合は，root 権限がないことも多い．

既存の監視手法では，監視対象ホストのプロセスとして起動する監視ソフトウェアや，VM として起動しデバッグを行う手法がある．また，緊急時，例えばカーネルパニックがおきた際はコアダンプの解析を行う．しかし，大量のコンピュータを管理するにあたり，それぞれのコンピュータに上記の対策を施すことは難しい．なぜなら，コンピュータによってオペレーティングシステムの設定が異なることがあるからである．

そこで本研究では，大量のコンピュータに対して，電源さえ入っていればオペレーティングシステムが停止していても動作するデバッグ環境である NetTLP[5] 上で，各コンピュータにおけるオペレーティングシステムのビルドコンフィグの差異を吸収する実装と評価を行った．

キーワード

OS

慶應義塾大学 環境情報学部

石川 達敬

Abstract Of Bachelor's Thesis　Academic Year 2019

Title

Summary

abst

Keywords

OS

Bachelor of Arts in Environment and Information Studies
Keio University

Tatsunori Ishikawa

目 次

図 目 次

表 目 次

第1章　序論

1.1　背景

コンピュータの管理者は，動作中，あるいはカーネルパニックなどによって停止したコンピュータの情報を監視・解析することが必要となる場面がある．動作中のコンピュータ自身に対しては，同一ホスト内の top コマンドや ps コマンドを用いて，プロセスの一覧を得たり，gdb コマンドを用いてプロセスをトレースし，プロセスの状態を把握する．ユーザー空間ではなくカーネルのデバッグしたい場合は，kdb と呼ばれるデバッガを，カーネルビルド時に有効にすることで，使用することができる．

論理的に停止したコンピュータに対しては，kdump と呼ばれる機構を通してメモリダンプを静的に解析し，原因の究明をする．また，状態を監視したいホストを物理的なマシンではなく，仮想マシンとして起動し，qemu や Xen などの基盤上で libvmi などを通して状態を解析する手法がすでに存在している．

上述した状況は，コンピュータの管理者，すなわち root 権限を保持している人にとって可能な手法である．

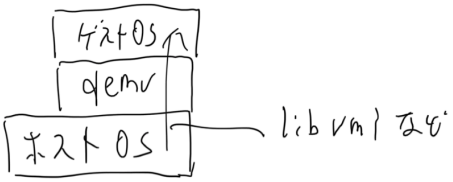
一方で，データセンター管理者など，大量の物理サーバーを保持し，顧客に貸し出している人の場合，上記の手法を使用することはできない．通常は，顧客の情報にアクセスすることはするべきではないが，例えば貸し出しているサーバーがマルウェアなどに感染するなどした場合に事業者としての責任として，原因究明や現状調査のために，解析する必要が出てくる可能性がある．

サーバーを貸し出している会社は，本来はセキュリティ対策として，サーバーを稼働しているオペレーティングシステム上に，セキュリティソフトを入れたいが，大量にあるコンピュータの全てにセキュリティソフトを入れることは容易ではない．当然，顧客から root パスワードを知らされることもないため，ログインをすることもできない．

1.2　課題

1.1 で述べたように，データセンターの管理者は，顧客に貸し出している物理的なコンピュータの内部の状態を知ることはできない．すなわち，死活監視として，ネットワーク越しの監視を行うことは可能であるが，コンピュータのオペレーティングシステムにおけるコンテキストを知ることはできない．オペレーティングシステムの内部で起こっていることは，当然，通常は知るべきではないが，マルウェアに感染した場合，意図しない挙動を起こした場合，あるいはカーネルパニックに陥った場合，これらの状態の時に，どの状態でも監視・解析を行うことができるツー

図 1.1: libvmi を用いる際のアーキテクチャ



ルが存在しない．

VM を用いた解析では，様々な解析手段がすでに豊富にあることは，1.1 で述べ，後述するとおり，VM を管理している物理的なコンピュータに以上が起きた場合に対処ができない．

大量の物理的なコンピュータに対する監視の場合，ネットワーク越しの死活監視，あるいはコンピュータの中でプロセスとしてセキュリティソフト，あるいは状態監視ツールを起動するほかない．この手法は，大量のコンピュータに適用するのは，現実的ではない．第一に顧客に貸し出すサーバーのリソースを微量でも使用してしまうという点．第二に，全てのサーバーにセットアップするのが大変だという点（加筆が必要）

また，このプロセスとして起動する方法は，物理的なコンピュータのオペレーティングシステムがカーネルパニックに陥った際，コアダンプの解析を行う他に解析手段が存在しない．コアダンプの設定を正しく行っていれば，静的ファイルを解析することは可能であるが，ストレージの不足など，不足の事態によって，ダンプを取ることができない可能性も存在する．

さらに，マルウェアなどに感染し，ログインをして解析することが危険にさらされる可能性もある場合，コンピュータにログインすることが推奨されない場合も存在する．

以上のことをまとめて，本研究における解決したい課題として，ネットワーク越しにある物理的なコンピュータに対して，電源さえ入っていればリアルタイムで安全に解析可能なオペレーティングシステムの監視・解析ツールが存在しないこと，と定義する．

1.3 目的

本研究では，大規模データセンターのような，大量かつ様々な環境の物理的なコンピュータを管理する現場において，root 権限がない中でネットワーク越しにあるコンピュータの状態を一元的に把握できることを示すことで，1.2 で述べた問題を解決することを目指す．

この章で述べた様々な環境とは，同じバージョンのオペレーティングシステムでもビルドする際の設定によって，挙動が変わるという意味で述べている．

本研究の実装によって，事前に与える情報が少ない中で，オペレーティングシステムのコンテキストを復元できることを示すために，タスクキューに乗っているプロセスの一覧を取得することを目指す．

1.4 本論文の構成

2 章では、既存の解析基盤や様々な解析手法と，基盤技術として使用することになる RDMA の既存の実装について述べる．

3 章では、本研究のアプローチとして，メモリからオペレーティングシステムのコンテキストの復元に必要な情報について述べる．

4 章では、本研究で実装したものについて述べる（加筆）

5 章では、本研究における評価として，Linux カーネルのバージョンのみが与えられた状態でプロセスリストの一覧を取得できることを示す．

6 章では、本研究に関する結論と，今後の課題について述べる．

第2章　関連技術

本章では，本研究における手法を選ぶに当たって，既存の基盤手法の比較と，基盤技術として使用する RDMA（Remote Direct Memory Address）に関して述べる．

2.1　オペレーティングシステム解析手段

本セクションでは，1 で述べた，既存のオペレーティングシステムおよびプロセスの解析技術について述べる．コアダンプを用いた静的解析や，kgdb，VM を用いた解析に関して述べた後，その手法の一つである libvmi について述べる．

2.1.1　コアダンプを用いた静的解析

コアダンプとは，カーネルクラッシュダンプとも呼称する [4] が，この技術は，オペレーティングシステムが何かしらの原因でパニックに陥った際に，停止した時点のメモリの情報を 2 次記憶装置に書き出し，あとで解析できるようにするための機構である．

Linux においては，`kdump` と呼ばれる機構を通して，メモリダンプを取得する．適切に設定をしておくことで，システムはパニックに陥ったのち，`kdump` を実行するためだけの緊急用のカーネルを起動し，メモリの内容を書き出していく．

ここで得られたファイルを，Volatility[3] のようなツールを用いて，オペレーティングシステムが停止する前にどのような状態にあったのかに関する解析を行う．

(1)　Volatility

Volatility の説明を書く．

2.1.2　kgdb

kgdb の説明

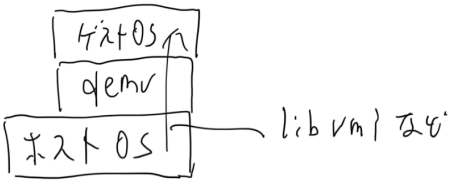
2.1.3 VM を用いた解析

VM を用いた解析では，監視したいホストを VM として起動することで監視を実現する手法である．

VM として起動する際に用いる技術としては，QEMU[2] がある．qemu とはコンピュータ全体をエミュレーションし，仮想マシンとしてオペレーティングシステムを起動するためのソフトウェアである．qemu ではプロセッサだけでなく，マウスやキーボードなどの周辺機器をエミュレートするため単体での使用も可能だが，近年では，Linux カーネルに実装されている仮想化モジュールである KVM[1] と組み合わせて使用することも多くなった．

この手法では，2.1 に示したように，ホスト OS の上で qemu を通してゲスト OS を実行する．

図 2.1: 監視対象ホストを VM として起動する場合



(1) libvmi

QEMU の上で実行する libvmi[6] というものがあり．これを使用した解析も可能である．

詳しく書く

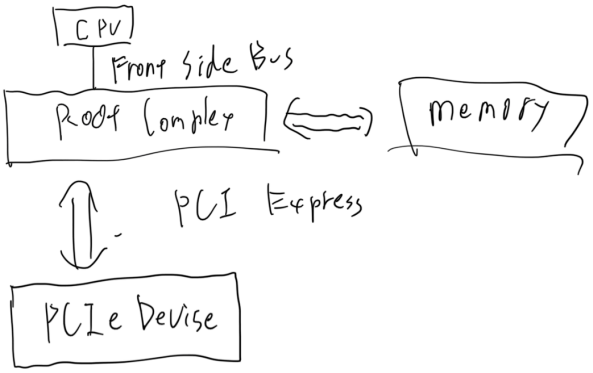
2.2 RDMA

ここに RDMA の説明をかく

2.2.1 Infiniband における RDMA 実装

制約がかなり厳しく，本研究の用途には適さないということを書く

図 2.2: PCI Express



第3章　アプローチ

1.3 で，本研究の目的を，動作中のコンピュータのメモリのダンプをリアルタイムで解析することで，コンピュータの状態をリモートホストから知ることができるようにする，と定義した．

そこで本章では，メモリのダンプをリアルタイムで取得．解析する上で前提となる情報と，この手法における課題について述べる．

3.1　オペレーティングシステムのコンテキスト

コンピュータの状態，すなわちオペレーティングシステムの動作中におけるコンピュータのコンテキストは，コンピュータ内部におけるレジスタの値および，内部から参照できる仮想アドレス空間上に保持されている．その例を下に示す．

あるプロセスを実行する際に，プロセッサはインストラクションポインタレジスタの命令を読み込み，逐次実行をしていく．call 命令などで別の関数を呼ぶ際には，その時点におけるインストラクションポインタレジスタの値をメモリ上に退避し，関数が終わった際に，呼び出し元に返るように設定されている．実行コードが整合性を保っているかは，実行可能ファイルを生成したコンパイラの責務なので，本論文では述べないが，プロセッサはプログラムの実行を行う際，レジスタの値を参照，退避，復帰，上書きさせることで，状態を保持，進行させていると言える．

プロセッサがレジスタの値を参照しそれを仕組みは，カーネルのコードを実行する際にも言える．ここでは，オペレーティングシステムから見たコンピュータの状態として，プロセスの切り替え処理，コンテキストスイッチにおける処理の流れを述べる．コンテキストスイッチとは，割り込み処理などによって定期的に呼ばれるプロセススケジューラーから呼ばれる機構である．この機構は，実行中のプロセスの状態，すなわち，各レジスタの値および仮想アドレス空間に関する情報などをカーネルが管理しているメモリ上にあるデータ構造の中に退避する．

本セクションのまとめとして，コンピュータの状態は，ある瞬間においてはレジスタの値であり，この状態を保存する際は，メモリ上にレジスタの値を退避させていることを述べた．

3.1.1　task_struct 構造体

3.1 でコンテキストスイッチにおいて，退避されるプロセスの情報は，対応したデータ構造に退避されると述べたが，この時に使用されるデータ構造が task_struct 構造体である．task_struct 構造体には，一つのプロセスの情報の全て（？）が格納されている．その中には，仮想アドレス空間に関する情報を保持する mm_struct 構造体を参照するフィールドも存在する．

コンテキストスイッチ時には，task_struct 構造体に保持されている情報をレジスタに復帰させることで，中断される直前の情報を復元している．

3.1.2　オペレーティングシステムのビルドにおけるコンフィグ

task_struct 構造体をはじめとして，Linux カーネルの変数や型，関数は，様々なアーキテクチャやカーネルコンフィグに対応するため，マクロによって分岐されている．この分岐が確定するのは，Linux カーネルをビルドするときであり，構造体のメンバへのアクセス，関数のアドレスなどはコンパイラが保証している．

実際のカーネルのバイナリは，vmlinux としてコンパイルされた後，strip され bzImage となる．ユーザーが作成したカーネルモジュールなどで関数を呼び出す際は，シンボルとアドレスの変換表である‘/boot/System.map‘を参照し，仮想アドレスを得たのち，実際にメモリにアクセスする際に物理メモリアドレスを算出しアクセスしている．

3.1.3　ホスト自身によるレジスタやシンボルの参照

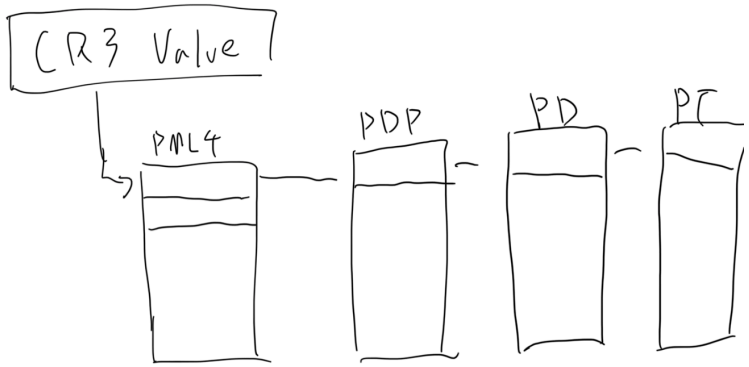
書き直し！！

3.1 で述べたように，オペレーティングシステムでは，レジスタの値などを退避する際，そのプロセッサ自身が’push’ 命令などを用いてメモリにアクセスできる

レジスタを参照して，かつ mmu すら自分で参照ができる．

CPU レジスタの現在の値は直接知ることができないため，例えばプロセスの一覧を取得したい場合は，コンテキストスイッチ時に退避された値を辿っていく必要がある．しかし，上述の通り task_struct はビルドされた際のカーネルコンフィグによって，どのメンバが先頭アドレスからどのオフセットに保持されているかは変動する．

図 3.1: Linux における mmu



3.2　本研究で保持する情報

これまでで述べたように，本研究では，メモリダンプを局所的に取得し，リアルタイムで解析することで，物理的に異なるコンピュータからオペレーティングシステムのコンテキストを復元することを目的と設定した．

3.3　なければならない情報

オペレーティングシステムの状態を保持しているものとして，3.1 で述べたようにレジスタがある．しかし，3.2 で述べたように，現在のレジスタの値は，メモリから知ることはできない．

また，退避された値は，メモリから読み出しを行うが，その際には，

第4章　実装

4.1　実装の概要

本研究では，RDMA を用いて，動作中のマシンのメモリの値を取得していくことで，リモートホストから監視対象ホストのオペレーティングシステムのコンテキストを復元していくことを目指す．この目的を実現するために，本研究では，NetTLP[5] を用いて実験を行う．

4.2　NetTLP

NetTLP の目的は，PCIe デバイスの開発プラットフォームである．

その機能の一つとして，DMA message と ethernet パケットを相互変換する機能がある．2 で述べたが，RDMA の Infiniband 実装は，制限が多い．（もう少し詳しく（3 章に詳しく書く））NetTLP における RDMA では，物理アドレスを指定することで，1Byte から 4096Byte までの任意のバイト数の値を取得することが可能である．また，NetTLP を用いた RDMA では，メインメモリの全メモリアドレスにアクセスすることが可能であり，アクセスできないメモリアドレスは存在しない，すなわち全メモリアドレス空間から値を取得することが可能となっている．

NetTLP は FPGA ボード上で動作するものであり，これを利用するためのインターフェースとして，libtlp が用意されている．libtlp では，RDMA を用いてメモリダンプを取得するためのインターフェースが関数として用意されている．この関数を含んだヘッダファイルを include し，プログラムから呼び出すことで，メモリアドレスの値が返ってくる．

用意されている関数は，`dma_read` 関数と `dma_write` 関数の二つである．`dma_read` 関数は，値を読みだすための関数であり，呼び出す際に読みたいメモリアドレスを渡す．`dma_write` 関数は，値を指定した物理アドレスに書き込むための関数であり，呼び出す際に，書き込みたいメモリアドレスと値を渡す．

本研究では，`dma_read` 関数のみを用いる．

4.2.1　NetTLP における process-list.c

このファイルでやっていることをかく．本研究の実装が，この process-list.c を拡張したものであるということを書く．

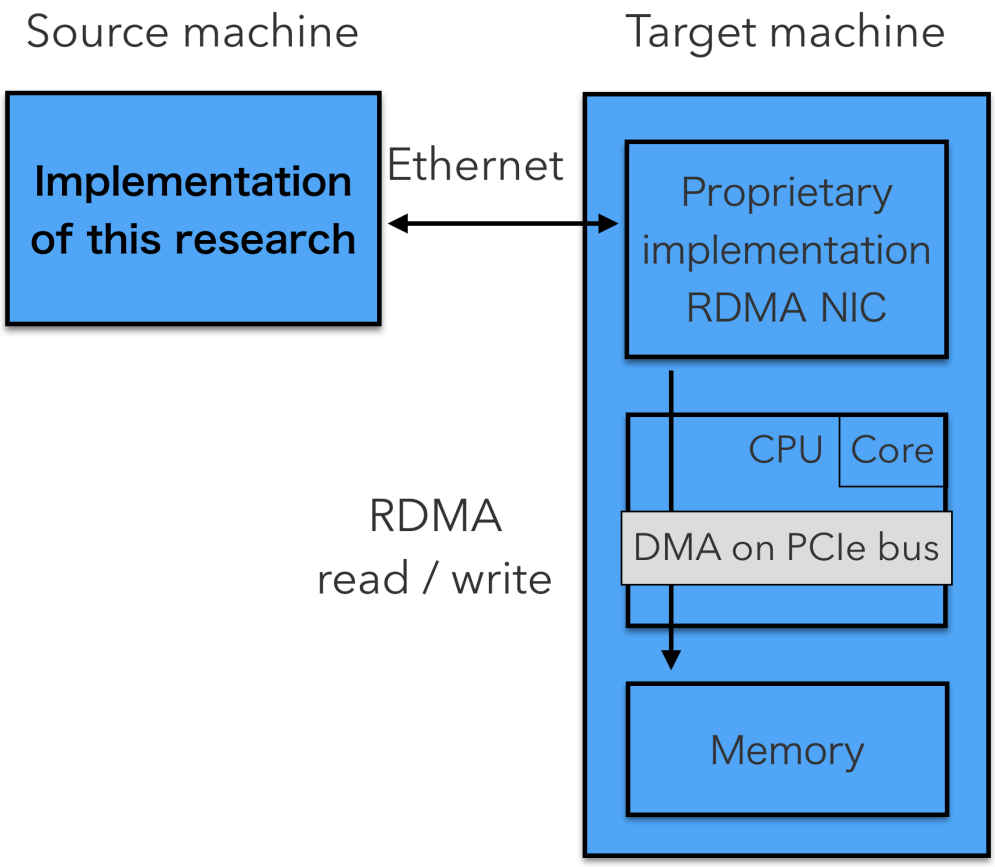
4.3 実験環境

本研究で実装を行う環境は，図 4.1 にあるように，NetTLP Adapter が書き込まれた FPGA が刺さった監視対象ホストと，本研究における実装を実行するホストの 2 台で構成する．

監視対象ホストは，Linux 4.15.0-72-generic の ubuntu であり，PCIe デバイスとして，NetTLP が書き込まれた FPGA ボードが刺さっている．本研究では，FPGA ボードとして，ザイリンクスのやつを使用している．(要加筆) また，この FPGA ボードは，ネットワークインターフェースでもあり，本研究の実験環境では，IP アドレスとして，192.168.10.1 を静的に振ってある．

実装を実行するホストは，Linux 4.19.0-6-amd64 の Debian buster であり，光ファイバーケーブル (名称はあとで修正) が刺さる NIC を刺している．以後，実装ホストと呼称する．この NIC には IP アドレスとして，192.168.10.3 を静的に振ってある．監視対象ホストに対して RDMA を実行する際は，`dma_read` 関数，あるいは `dma_wirte` 関数を通して 192.168.10.1 に対して IP パケットを送信する．

図 4.1: 全体



4.4 実装の前提情報

本研究では，3 章で述べたように，動作中のコンピュータのメインメモリの値を読むことによって監視対象ホストにおけるオペレーティングシステムのコンテキストを復元することを目的としている．その手法として，RDMA NIC を物理的に設置することで，この目的を達成することを試みている．

そこで，本研究の実装側のホストに与える情報を少なくすることが必要となる．本セクションでは，本研究の実験において，実装を実行するホストが持っている情報と，初期段階では持っていないが解析の結果導き出す情報を分類する．

3 章で述べたように，オペレーティングシステムのコンテキストの復元，その中でも Linux においてプロセス情報の一覧を出すために必要な情報は以下の三点である．

一点目は，init$_t ask$ という，pid が 0 のプロセスの $task\_struct$ 構造体の開始アドレスである．$init_t ask$ はコンピュータが起動する際に最初に実行されるプロセスであり，全てのプロセスは親プロセスを辿っていくことで，このプロセスにたどり着くことができる．この情報は，実験に際して実装を実行するホストは，知らないこととする．（いいのか？）

二点目は，task_struct 構造体の各フィールドの有無である．Linux カーネルでは，ビルドする際に，数千に及ぶ設定を記述し，マクロとして設定される．この設定，kconfig によって task_struct は，どのフィールドを有効にするか，マクロとして定義された構造体の実体は何になるのか，などが決まる．kconfig の結果によって，フィールドが存在するか否か，またそのフィールドが先頭アドレスからどのくらいのオフセットを持った状態で保持されているかが決まる．すなわち，kconfig の情報によって，task_struct のサイズや各フィールドの先頭アドレスからのオフセットが確定する．この kconfig に関する情報は実験に際して実装を実行するホストは知らないこととする．

三点目は，Linux カーネルのバージョンに関する情報である．本研究では，実験する際に，監視対象ホストのカーネルのバージョンと同じソースコードを使用した上で実験を行う．当然，Linux カーネルのバージョンに関する情報は知っている必要がある．Linux カーネルのバージョンは，実装ホストは持っている情報とする．3.3 で述べたもののうち，Linux カーネルのバージョンは通知することとする

また，KASLR（kernel address space layout randomization）を無効にしてある．理由については，4.4.1 にて述べる．

4.4.1 KASLR

KASLR の簡単な説明と，無効にする理由と無効にしても良い理由を書く．

KASLR とは，kernel address space layout randomization の略であり，カーネル領域における仮想アドレス空間のランダム化に関する仕組みのことである．基本的な仕組みは，カーネルに限らず，プロセスの仮想アドレス空間のランダム化を実現する，ASLR(address space layout randomization) がある．ASLR では，実行ファイル内に存在するコードセグメントの仮想アドレ

スを，プロセスの実行のたびにランダムにする．

ASLR が無効な状態でプロセスを実行すると，コードセグメントなどが常に一定の仮想アドレス空間にロードされるため，攻撃者にとって，仮想アドレスの推測が容易になってしまう．これを防ぐための機構が ASLR である．

KASLR では，この考え方をカーネルの起動時にも適用するようにしたものである．

4.5 実装の全体

ダンプしてくるということを書く．物理アドレスのマッピングに関しても書く

実験における第一段階として，3.3 で述べたように，監視対象ホストのカーネルコンフィグおよび `init_task` の先頭アドレスの仮想アドレスを知ることを目指す．そこで，本研究では，この情報をメモリ上から探す．4.6 で述べる実装では，取得できるメモリダンプを全て取得し，解析する手法に関して述べる．

第二段階として，与えられた Linux カーネルのバージョンのソースコードより，カーネルコンフィグの一覧を抽出し，その文字列を取得したメモリダンプから文字列探索をする．文字列探索の結果取得したカーネルコンフィグの値をパースし，メモリ上から監視対象ホストのカーネルコンフィグを復元する．4.7 で述べる実装では，カーネルコンフィグを復元する際の実装の詳細に関して記述する．

第三段階として，収集したカーネルコンフィグを元に手元のコンピュータで Linux カーネルのソースコードに対してプリプロセスの処理を行い，task_struct 型を確定する．また，**??**で述べた，監視対象ホストで動いているプロセスの一覧情報を取得するためのさらに，ソースコード上にある `__phys_addr` 関数の実体を収集する．

最後に，この工程で得られた情報をもとに，libtlp で提唱されている手法を用いて，プロセスの一覧を正しく取得できることを確認する．

4.6 mem_dump.c

第一の工程として，メモリの全ての情報を取得する．ソースコードは以下である．この実装を実装ホストで実行し，出力結果をファイルに格納する．この実装では，libtlp を通して，監視対象ホストのメモリを全探索する．この実装の実行には長い時間（何分？）かかるため，アトミックな情報ではない．そのため，ここで取得したメモリダンプは，解析には使えない．ここで取得したメモリダンプは，System.map のうち，`init_task` が配置されている仮想アドレス空間に関する情報および，Linux カーネル 4.15.0 におけるカーネルコンフィグに関する情報を収集するためのものである．

実行方法

```
./dump_mem > dump
```

4.7 カーネルコンフィグの復元

Linux カーネル 4.15.0 におけるカーネルコンフィグの一覧は，下に示す通りである（あとではるかも）

これらのコンフィグに関する情報を以下のスクリプトで読み出す．

カーネルコンフィグには，各設定項目に対する値として，y,m や文字列，数値などがあり，設定しない項目については，その行がコメント行になるのに加えて，`is not set` という文言が付け足される．これらの特徴を踏まえ，本研究では，restore$_k$*config.py* というスクリプトを *Python* を用いて実装した．このスクリプトでは，得られたメモリダンプから，*strings* コマンドを用いて文字列を抽出し，そこから *kconfig* の特徴である，*CONFIG* という文字列を含む行を *grep* コマンドを用いて抽出する．実行するシェルスクリプトは以下である．

strings

```
strings dump | grep CONFIG > str_list
```

生成されたファイルに対して，上述したスクリプトを実行し，ファイルに書き出す．ここでは書き出すファイル名を restored$_k$*config* とする．

search config script

```
import sys

configs = [
    "CONFIG_64BIT", "CONFIG_X86_64", "CONFIG_X86",
    "CONFIG_INSTRUCTION_DECODER", "CONFIG_OUTPUT_FORMAT",
    "CONFIG_ARCH_DEFCONFIG", "CONFIG_LOCKDEP_SUPPORT",
    # 省略
    "CONFIG_ARCH_HAS_PMEM_API", "CONFIG_ARCH_HAS_UACCESS_FLUSHCACHE",
    "CONFIG_SBITMAP", "CONFIG_PARMAN", "CONFIG_STRING_SELFTEST"
]

# 有効な文字列が見つかった場合は 1 を返す
def classification(l, s):

    if "#if" in l or "#endif" in l:
        # sys.stderr.write("No!! -> Macro, "+l)
        return 0

    if l[:3] != "CON" and l[:3] != "# C":
        # sys.stderr.write("No!! -> NOT CONFIG, "+l)
        return 0

    if s + "=y" in l:
        print(s + "=y")
        return 1
    elif s + "=m" in l:
        print(s + "=m")
        return 1
    elif s + " is not set" in l:
        print("# " + s + " is not set")
        return 1
    elif s + '=' in l:
        print(l)
        return 1
    else:
        # sys.stderr.write("No!! -> No match, "+l)
        return 0
```

search config script2

```
def search_config_str(file_name, s):
    ld = open(file_name)
    lines = ld.readlines()
    ld.close()

    for line in lines:
        if line.find(s) >= 0:
            if classification(line[:-1], s):
                return 1
    return 0


def search(file_name):
    for s in configs:
        # print("Searching "+s+" .....")
        if not search_config_str(file_name, s):
            # print("# Cannot find " + s)
            sys.stderr.write("# Cannot find " + s)


def usage():
    print("usage: python find_kconfig.py path/to/str_list")


def main():
    args = sys.argv
    if len(args) < 2:
        usage()
        return 0

    file_name = args[1]
    search(file_name)


if __name__ == "__main__":
    main()
```

上述した処理によって得られた restored$_{k}config$ というファイルを，後述するビルド時にコンフィグとして利用する．

4.8 Linux カーネルをプリプロセッサに通す

この工程では，収集したカーネルコンフィグを元に手元のコンピュータで Linux カーネルのソースコードに対してプリプロセスの処理を行い，task_struct 型，および__phys_addr 関数など，process-list.c の影響のあるソースコードを確定する．

また，pid 0 のプロセスの task_struct 構造体の先頭アドレスを知るため，またそれぞれのフィールドの先頭アドレスからのオフセットを確定させるため，オフセットを得る処理を施す．

まずは，事前に知らされた情報である Linux カーネルのバージョンより，適合したカーネルを取得する．このソースコードに対して，4.7 で生成したファイルをコンフィグとして埋め込む．

build Linux kernel

```
cd /path/to/linux-source-4.15.0
cp /path/to/restored_kconfig .config
```

また，この工程では，カーネルビルド時における task_struct 型をバイナリではなくテキストファイルとして取得する必要があるため，マクロを適用した直後の状態，すなわちプリプロセッサに通した直後の状態を保存するため，ビルド時の設定に変更を加える．Linux カーネルにおいては，ビルド時に Makefile を使用するため，このファイルを編集する．例として，Linux カーネル，バージョン 4.15.0-74-generic においては，Makefile の 447 行目付近に，-save-temps オプションを以下のような形で設定する．

-save-temps オプションの設定・変更前

```
KBUILD_AFLAGS   := -D__ASSEMBLY__
KBUILD_CFLAGS   := -Wall -Wundef -Wstrict-prototypes -Wno-trigraphs \
        -fno-strict-aliasing -fno-common -fshort-wchar \
        -Werror-implicit-function-declaration \
        -Wno-format-security \
        -std=gnu89
KBUILD_CPPFLAGS := -D__KERNEL__
KBUILD_AFLAGS_KERNEL :=
KBUILD_CFLAGS_KERNEL :=
KBUILD_AFLAGS_MODULE  := -DMODULE
KBUILD_CFLAGS_MODULE  := -DMODULE
KBUILD_LDFLAGS_MODULE := -T $(srctree)/scripts/module-common.lds
```

-save-temps オプションの設定・変更後

```
KBUILD_AFLAGS   := -D__ASSEMBLY__
KBUILD_CFLAGS   := -Wall -Wundef -Wstrict-prototypes -Wno-trigraphs \
        -fno-strict-aliasing -fno-common -fshort-wchar \
        -Werror-implicit-function-declaration \
        -Wno-format-security \
        -std=gnu89

KBUILD_CFLAGS += -save-temps=obj

KBUILD_CPPFLAGS := -D__KERNEL__
KBUILD_AFLAGS_KERNEL :=
KBUILD_CFLAGS_KERNEL :=
KBUILD_AFLAGS_MODULE  := -DMODULE
KBUILD_CFLAGS_MODULE  := -DMODULE
KBUILD_LDFLAGS_MODULE := -T $(srctree)/scripts/module-common.lds
```

設定ファイルを書き終わったらビルドを行う．

ビルド

```
make -j10
```

4.9 task_struct 構造体の確定

本セクションでは，4.8 で述べた工程を経た結果生成された中間ファイルから，task_struct 構造体を導出し，プロセス情報一覧の表示に必要なフィールドのオフセットを求める．

Linux カーネルのビルドが完了すると，上述した Makefile の KBUILD$_CFLAGS$ の設定によって，中間ファイルを含む巨大なディレクトリおよび，$vmlinux, bzImage$ が作成される．本研究では，このビルドされた $bzImage$ は使用しない．

ビルドの際に，プリプロセッサの出力を残したことで，ソースコード中の全てのマクロおよび include されたファイルが展開された状態のソースコードがファイルとして残っている．例として/kernel/pid.c をあげると，このファイルでは，task_struct 構造体を呼び出している箇所があるが，このファイルをプリプロセッサに通すことで，pid.i が作成される．pid.i を通して見てみると，task_struct 構造体が全て展開され，そこから参照される全ての構造体や typedef の情報がソースコード上にあることがわかる．

この中間ファイルから，task_struct およびそこから参照される全ての要素を抽出し，以下のソースコードの `struct task_struct{};` と書かれている部分に記述する．このファイルをビルドす

ることで，task_struct 構造体の各フィールドのオフセットを導出する．

$print_offset.c$ の実行結果は以下の通りである．

ここで得られたオフセットを用いて，後述する 4.9.1 にて $init_task$ の開始アドレスを求める．

4.9.1 $init_task$ の開始アドレスの算出

本セクションでは，プロセス ID として 0 を持つプロセスである，$init_task$ の先頭アドレスを算出する．

$init_task$ の開始アドレスは，監視対象ホストの，$/proc/kallsyms$ に記述されているが，その開始アドレスは本研究の実験環境においては，実装を実行するホストは情報として持っていない．そのため，後述する手法を用いてその開始アドレスを算出する．

4.6 では，ネットワーク越しに，監視対象ホストのメモリダンプを取得する工程について記述した．このメモリダンプからプロセス ID 0 をもつ $init_task$ を探す．$init_task$ は $task\_struct$ 構造体であるため，メモリダンプの中から，$init_task$ に特有の文字列をなどを探し，それを目印として $init_task$ の先頭アドレスを算出する．

本研究においては，task_struct 構造体の，comm フィールドの値に着目した．comm フィールドには，プロセスに関する情報のうち，実行可能ファイルの名前が 16Byte で記載されている．$init_task$ における $comm$ フィールドの値は，$swapper/0$ であるため，この値をメモリダンプから，以下のスクリプトを用いて検索を行う．

find swapper/0

```
xxd dump | grep swapper/0
```

この値から，**??**structsection:define$_task_struct$ た，$comm$ フィールドの値を引き，そこからさらに，ブートローダの使用領域である $128KB$ を足すことで，$init_task$ の先頭アドレスを算出する．

4.10 プロセス一覧の表示

4.8 で得ることができたソースをもとに，process-list を改造したものに関する説明をここに書く（もう動いてはいるので，整理して書く．）

以上の実装により得られた値を用いて，**??**で述べた process-list.c のうち，監視対象ホストの環境に依存した部分を書き換えることで，プロセスの一覧を取得する．

4.10.1 環境に依存するパラメータ

本セクションでは，プロセスの一覧を得る上で，マシンごとに異なる設定を述べる．

一点目として，カーネル空間に仮想アドレスにおける仮想アドレスから物理アドレスへ変換する際に使用する関数の実体が異なる．/proc/kallsyms に書いてある値をはじめとして，子プロセスの開始アドレスを格納しているフィールドには，カーネル空間における仮想アドレスが格納されているが，本研究においてメモリアドレスを指定する際には，物理アドレスを指定する必要がある．Linux カーネル 4.15.0 においては，変換に用いる関数およびその中で使用されているシンボルは，CONFIG$_{D}EBUG_{V}IRTUAL$ という設定や，64*bit* かどうかを示す値であり，この値はソースコードからマクロを辿っていくことで知ることが可能である．本研究では，4.7 にて述べたように，復元したマクロの値を参照しつつ，この関数の実体を確定させる．

二点目として，監視対象ホストの上における task_struct 構造体におけるオフセットの値である．この値に関しては，4.9 で求めたため，その値を以下の 6 行に記載する．

macros

```
#define OFFSET_HEAD_STATE 16
#define OFFSET_HEAD_PID 2216
#define OFFSET_HEAD_CHILDREN 2248
#define OFFSET_HEAD_SIBLING 2264
#define OFFSET_HEAD_COMM 2640
#define OFFSET_HEAD_REAL_PARENT 2232
```

4.11 実装のまとめ

本章では，監視対象ホストに関する情報として，動作している Linux カーネルのバージョンのみを実装を実行するホストに与えた．その上で，RDMA の NetTLP 実装を用いてメモリダンプを取得し解析を行うことで，カーネルコンフィグをはじめとした，プロセス一覧の取得に必要な情報を収集するための実装について述べた．

第5章　評価

本章では，本研究における実装によって，正しく監視対象ホストの状態を取得できているかどうかを評価とする．また，hoge，fuga な時にもその評価が正しくできているかを確認する．

5.1　評価手法

評価手法として，カーネルのバージョンのみわかる状態から，正しく ps aux と同じような出力を得られるかどうか，実験用に起動したプロセスを，本研究の実装上から確認できるかどうかを評価とする．また，実験の最中に導出した値が実際のホストにおいて正しいかどうかを確認し，それを評価とする．

プロセスとして監視を行う手法と，本研究の実装を，通常稼働中とカーネルパニック発生時における実行の可否について述べる．

5.2　実験環境

本研究では，以下の環境で実験を行う．

表 5.1: 実装を実行するホスト

Linux カーネルのバージョン	Linux 4.19.0-6-amd64
ディストリビューション	Debian buster 10.2

表 5.2: 監視対象ホスト

Linux カーネルのバージョン	Linux 4.15.0-74-generic
ディストリビューション	Ubuntu 18.04.3 LTS (Bionic Beaver)

5.3　評価手順

評価手順として，4 章で述べた実装を用いて，実際に全ての工程を，手順に沿って実行していく．

5.3.1 前提

前述したように，本研究の実験においては，実装を実行するホストは，監視対象ホストに関して，Linux カーネルのバージョンのみを情報として保持する．

5.3.2 メモリダンプの取得

4.6 で述べた実装である，dump$_mem$ を用いて，メモリダンプを取得する．

実行方法

```
./dump_mem > dump
```

このファイルを実行すると，搭載している物理メモリの大きさに等しい，8GB のファイルが作成される．

このファイルを以後，メモリダンプと呼ぶ．

5.3.3 カーネルコンフィグを復元する

取得してきたメモリダンプに対して，4.7 で述べたように，処理を施し，ビルド時のカーネルコンフィグを復元する．

strings

```
strings dump | grep CONFIG > str_list
```

ここで生成された resotred$_k config$ を，実装を実行するホストでカーネルをビルドする際に，$.config$ としてそのまま用いる．

5.3.4 復元したカーネルをプリプロセッサに通す

4.8 で述べたように，5.3.3 の結果得られたカーネルコンフィグを用いて，カーネルのビルドを実装を実行するホストで行う．

build Linux kernel

```
cd /path/to/linux-source-4.15.0
cp /path/to/restored_kconfig .config
```

その際に，4.8 で述べたように，プリプロセッサによる処理である中間ファイルを残す設定とするため，Makefile に変更を加える．本研究では，監視対象ホストのバージョンは，Linux 4.15.0-74-generic でありその変更は，4.8 で述べたものと同じである．

変更したのちに，以下のコマンドを実行し，ビルドを開始する．

ビルド

```
make -j10
```

ビルドが終了すると，ソースコードが中間ファイルの生成によって以下のようなサイズとなる．

ビルド

```
$ du -shc linux-source-4.15.0
64G linux-source-4.15.0
64G total
```

5.3.5 実行環境における $\mathbf{init}_{t}ask$ の先頭アドレス

4.9 で述べた内容に基づいて，作成した $\mathrm{print}_{o}ffset$ を実行した結果は以下となった．

名前考える

```
$ ./print_offset_restore
# task_struct size: 9088

# state: 16
# pid: 2216
# children: 2248
# sibling: 2264
# comm: 2640
# real_parent: 2232
```

この結果をもとに，5.3.5 にて，$\mathrm{init}_{t}ask$ の先頭アドレスを算出する．

5.3.2 で取得したメモリダンプから，4.9.1 で述べたように，swapper/0 という文字列を以下のコマンドで検索を行う．

find swapper/0

```
$ xxd dump | grep swapper/0
# 023f3ed0: 7377 6170 7065 722f 3000 0000 0000 0000  swapper/0.......
# e09f3ed0: 7377 6170 7065 722f 3000 0000 0000 0000  swapper/0.......
```

以上の結果となった．このうち一つ目の値を取り出すと，023f3ed0 であるが，4.9.1 に倣って，0x023f3ed0 の 10 進表記である 37699280 から，comm フィールドのオフセットである 2640 を減算し，128KB のバイト数である 131072 を加算する．その結果，得られた値は，37699280-2640+131072=37827712 となる．この値は物理アドレスであるため，これをカーネルの仮想アドレスに変換する．

Linux カーネルで物理アドレスが 0 からのストレートマップとなるのは，上述のソースコード内における三項演算子のうち，条件式が真となる場合である．この関数から，37827712 の 16 進数表記である 0x2413480 という結果が帰る場合は条件式が真となる場合であるため，カーネル空間の仮想アドレスにおいて，37827712 という物理アドレスに対応する仮想アドレスは， 0x2413480 + phys_base + 0xffffffff80000000 の結果である 0xffffffff82413480 となり，これが本研究における監視対象ホストの init$_task$ の仮想アドレスと推定する．また，この値を $process - list.c$ の引数として使用する．

5.3.6 process-list.c の実行

5.3.5 で導いた値を引数として，以下のようにコマンドを実行する．実行結果については，5.4.2 で述べる．

process-list.c の実行

```
./process-list 0xFFFFFFFF82413480
```

5.4 評価

5.4.1 値が正しいこと

5.3 における評価手順において，導出した値として，init$_task$ の仮想アドレスが正しいかどうかを評価する．評価手法としては，実験の際に導いた $0xffffffff$82413480 という値が監視対象ホストの値と等しいかどうかを，監視対象ホストの $kallsyms$ を参照することで比較する．

比較結果は以下であり，導出した 0xffffffff82413480 という値が正しい値であることを示した．

kallsyms の出力

```
$ sudo cat kallsyms | grep "D init_task"
# ffffffff82413480 D init_task
```

5.4.2 通常稼働中における評価

5.3 によって求めた値を用いて，process-list.c を実行する．実行結果は以下である．

process-list の出力

```
$ ./process-list 0xFFFFFFFF82413480

init_vm_addr: 0xffffffff82413480
PhyAddr             PID STAT COMMAND
0x00000002413480      0 R: swapper/0
0x000002361f0000      1 S: systemd
0x0000022ea616c0    287 S: systemd-journal
0x0000022da0ad80    297 S: blkmapd
0x0000022e232d80    308 S: systemd-udevd
0x000002333c8000    527 S: systemd-timesyn
0x000002333cc440    530 S: rpcbind
0x00000233d72d80    534 S: cron
0x00000233d70000    536 S: atd
0x0000022e732d80    545 S: rsyslogd
0x0000022dbfad80    556 S: irqbalance
0x0000022dbf96c0    561 S: accounts-daemon
0x0000022dbfdb00    569 S: dbus-daemon
0x0000022f752d80    586 S: wpa_supplicant
0x0000022f754440    589 S: systemd-logind
0x0000022ea40000    592 S: networkd-dispat
0x0000022f7e0000    607 S: polkitd
0x0000022f750000    634 S: systemd-resolve
0x0000022da08000    663 S: dhclient
0x00000235378000    777 S: nmbd
0x00000234bac440    781 S: unattended-upgr
0x00000234baad80    783 S: sshd
0x00000234ba2d80   3009 S: sshd
0x0000022e735b00   3142 S: sshd
0x00000234ba16c0   3143 S: zsh
0x00000234ba8000    787 S: agetty
0x00000234ba0000    819 S: smbd
0x00000234452d80    821 S: smbd-notifyd
0x000002344516c0    822 S: cleanupd
0x00000234450000    823 S: lpqd
0x0000022f7e4440   3032 S: systemd
0x0000022f7516c0   3033 S: (sd-pam)
0x000002361f5b00      2 S: kthreadd
0x000002361f16c0      4 D: kworker/0:0H
0x0000023622ad80      6 D: mm_percpu_wq
0x000002362296c0      7 S: ksoftirqd/0
0x0000023622c440      8 D: rcu_sched
0x00000236228000      9 D: rcu_bh
0x0000023622db00     10 S: migration/0
0x00000236254440     11 S: watchdog/0
```

監視対象ホストにおける ps コマンドの結果

```
$ ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.0  0.1 225484  9196 ?        Ss   Jan27   0:14 /sbin/init nopti nospectr
root          2  0.0  0.0      0     0 ?        S    Jan27   0:00 [kthreadd]
root          4  0.0  0.0      0     0 ?        I<   Jan27   0:00 [kworker/0:0H]
root          6  0.0  0.0      0     0 ?        I<   Jan27   0:00 [mm_percpu_wq]
root          7  0.0  0.0      0     0 ?        S    Jan27   0:00 [ksoftirqd/0]
root          8  0.0  0.0      0     0 ?        I    Jan27   0:02 [rcu_sched]
root          9  0.0  0.0      0     0 ?        I    Jan27   0:00 [rcu_bh]
root         10  0.0  0.0      0     0 ?        S    Jan27   0:00 [migration/0]
root         11  0.0  0.0      0     0 ?        S    Jan27   0:00 [watchdog/0]
root         12  0.0  0.0      0     0 ?        S    Jan27   0:00 [cpuhp/0]
root         13  0.0  0.0      0     0 ?        S    Jan27   0:00 [cpuhp/1]
root         14  0.0  0.0      0     0 ?        S    Jan27   0:00 [watchdog/1]
root         15  0.0  0.0      0     0 ?        S    Jan27   0:00 [migration/1]
root         16  0.0  0.0      0     0 ?        S    Jan27   0:00 [ksoftirqd/1]
root         18  0.0  0.0      0     0 ?        I<   Jan27   0:00 [kworker/1:0H]
root         19  0.0  0.0      0     0 ?        S    Jan27   0:00 [cpuhp/2]
root         20  0.0  0.0      0     0 ?        S    Jan27   0:00 [watchdog/2]
root         21  0.0  0.0      0     0 ?        S    Jan27   0:00 [migration/2]
root         22  0.0  0.0      0     0 ?        S    Jan27   0:00 [ksoftirqd/2]
root         24  0.0  0.0      0     0 ?        I<   Jan27   0:00 [kworker/2:0H]
root         25  0.0  0.0      0     0 ?        S    Jan27   0:00 [cpuhp/3]
root         26  0.0  0.0      0     0 ?        S    Jan27   0:00 [watchdog/3]
root         27  0.0  0.0      0     0 ?        S    Jan27   0:00 [migration/3]
root         28  0.0  0.0      0     0 ?        S    Jan27   0:00 [ksoftirqd/3]
root         30  0.0  0.0      0     0 ?        I<   Jan27   0:00 [kworker/3:0H]
root         31  0.0  0.0      0     0 ?        S    Jan27   0:00 [kdevtmpfs]
root         32  0.0  0.0      0     0 ?        I<   Jan27   0:00 [netns]
root         33  0.0  0.0      0     0 ?        S    Jan27   0:00 [rcu_tasks_kthre]
root         34  0.0  0.0      0     0 ?        S    Jan27   0:00 [kauditd]
root         35  0.0  0.0      0     0 ?        I    Jan27   0:00 [kworker/0:1]
root         37  0.0  0.0      0     0 ?        S    Jan27   0:00 [khungtaskd]
root         38  0.0  0.0      0     0 ?        S    Jan27   0:00 [oom_reaper]
root         39  0.0  0.0      0     0 ?        I<   Jan27   0:00 [writeback]
root         40  0.0  0.0      0     0 ?        S    Jan27   0:00 [kcompactd0]
root         41  0.0  0.0      0     0 ?        SN   Jan27   0:00 [ksmd]
root         42  0.0  0.0      0     0 ?        SN   Jan27   0:00 [khugepaged]
root         43  0.0  0.0      0     0 ?        I<   Jan27   0:00 [crypto]
root         44  0.0  0.0      0     0 ?        I<   Jan27   0:00 [kintegrityd]
root         45  0.0  0.0      0     0 ?        I<   Jan27   0:00 [kblockd]
root         46  0.3  0.0      0     0 ?        I    Jan27   2:13 [kworker/2:1]
root         47  0.0  0.0      0     0 ?        I    Jan27   0:00 [kworker/3:1]
root         48  0.0  0.0      0     0 ?        I<   Jan27   0:00 [ata_sff]
```

(1)　特定のプロセス名の取得

以上の結果に加えて，ユーザーが独自に起動したプロセスの情報を取得できているかを下に示す．評価としては，user という無限ループするのみのユーザープロセスを起動し，その情報を取得する．

以下に実行結果を示す．

process-list から user というプロセスを検索

```
./process-list 0xFFFFFFFF82413480 | grep user
0x00000234ba96c0   4189 S: user
```

監視対象ホストの ps コマンドから user というプロセスを検索

```
ps aux | grep "user"
tatsu     3032  0.0  0.0  76648  7624 ?        Ss   02:47   0:00 /lib/systemd/systemd --us
tatsu     4189  0.0  0.0   4508   808 pts/1    S+   03:51   0:00 ./user
tatsu     4207  0.0  0.0  15452  1004 pts/0    S+   03:52   0:00 grep --color=auto --exclu
```

プロセス ID として 4189 を持つプロセスがどちらにも存在しているため，正しくプロセスの情報を取得できていることを示せた．

5.4.3　カーネルパニック発生時における評価

カーネルパニック発生時は既存の手法，プロセスとして起動する方法はだめだが，本研究における実装では問題なく動作することを示す．

5.5　評価のまとめ

未評価

第6章　まとめと結論

6.1　まとめ

各章の振り返り

本論文の各章を振り返る

2 章では、本研究で使用する RDMA を使う理由について述べた．

3 章では、メモリの情報からオペレーティングシステムのコンテキストを復元するということについて述べた．その上で，メモリからどのような情報を探すことで，オペレーティングシステム，コンピュータの状態を復元することができるのかという点について述べた．

4 章では、本研究で実装したものについて述べた（加筆）

5 章では、本研究における評価として，Linux カーネルのバージョンのみが与えられた状態でプロセスリストの一覧を取得できることを示した．

6.2　結論

結論は俺の知見，考察．結局こうだったを述べる．こういうところでは使えた，こういうところではダメだった．

本論文の結論として，4 で述べたような実装を用いることで，目的を達成することができた．（加筆）

6.3　今後の課題

特にカーネルバージョンの特定と KASLR のバイパスについて書く．

6.3.1　KASLR

KASLR は本来，ASLR として実装されたものであり，それのカーネルバージョンである．内部からの攻撃に向けた防御策であるため，この研究の手法であればバイパスは可能であると考えられる．なぜなら，メモリ保護機能，すなわちアクセスできるメモリアドレスの制限を受けないレイヤーで動作する環境だから．

6.3.2　セキュリティ的な課題

課題

謝辞

アドバイスをくれた全員に感謝

参考文献

[1] KVM. https://www.linux-kvm.org/page/Main_Page.

[2] QEMU. https://www.qemu.org/.

[3] The Volatility Foundation - Open Source Memory Forensics. https://www.volatilityfoundation.org/.

[4] 第 7 章 カーネルクラッシュダンプガイド Red Hat Enterprise Linux 7 — Red Hat Customer Portal. https://access.redhat.com/documentation/ja-jp/red_hat_enterprise_linux/7/html/kernel_administration_guide/kernel_crash_dump_guide.

[5] Nettlp: A development platform for pcie devices in software interacting with hardware. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA, February 2020. USENIX Association.

[6] Bryan D. Payne. Libvmi, version 00, 9 2011.