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，大量の物理的なコンピュータを管理するにあたり，問題に対処するための様々な解決策を導入することは難しい．

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 がプログラムされた FPGA ボードを監視対象ホストに物理的に設置するだけで，NetTLP に実装された RDMA の実装を用いてメモリ探索を行い，動作中のネットワーク越しにある物理的なコンピュータの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する．

実装および評価として，限られた情報の中で，ネットワーク越しにある 64bit Linux の監視対象ホストのプロセス一覧を，自ホストから取得し，復元，出力できることを示す．

RDMA を用いたメモリ探索でプロセス一覧を復元するに際し，本研究ではプロセス ID として 0 を持つプロセスである init_task を起点として探索を行う．init_task は task_struct 構造体を保持しているが，監視対象ホストで動作中の task_struct 構造体の型情報はカーネルコンフィグから推測するほかない．本研究では，task_struct 構造体の型情報の取得，すなわち各フィールドのオフセットの算出を，実装したプログラムを実行するホストにてアトミックではないメモリダンプを解析しカーネルコンフィグの値を収集，再度ビルドすることで達成している．

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

Managers of a large number of physical computers, such as a large data center, are faced with a situation in which computers are analyzed in addition to normal surveillance and investigation of causes in an emergency. However, if you rent a computer to a customer, you often do not have root privileges.

Existing monitoring methods include monitoring software that starts as a process on the monitored host and a method that starts and debugs as a VM. In an emergency, for example, when a kernel panic occurs, the core dump is analyzed.

However, in managing a large number of physical computers, it is difficult to introduce various solutions to address the problem.

Therefore, in this research, under the NetTLP environment, which is a debugging environment that operates even if the operating system is stopped as long as the power is on, for a large number of computers, The purpose of this study is to perform a memory search using the implementation of RDMA implemented in NetTLP and to restore the context of the operating system of a physical computer over a running network.

As an implementation and evaluation, we show that, with limited information, a process list of the monitored host of 64bit Linux over the network can be obtained from the host, restored, and output.

When restoring a process list by memory search using RDMA, in this research, search is performed starting from init_task, which is a process having a process ID of 0. init_task holds a task_struct structure, but the type information of the task_struct structure running on the monitored host must be inferred from the kernel config. In this research, acquisition of type information of task_struct structure, that is, calculation of offset of each field, This is achieved by analyzing the non-atomic memory dump on the host executing the implementation, collecting the kernel configuration values, and rebuilding.

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．

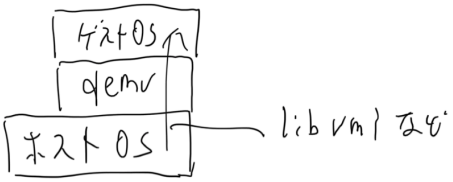
一方で，データセンター管理者など，大量の物理サーバーを保持し，顧客に貸し出している人の場合，上記の手法を使用することはできない．通常は，顧客の情報にアクセスすることはするべきではないが，例えば貸し出しているサーバーがマルウェアなどに感染するなどした場合に事業者としての責任として，原因究明や現状調査のために，解析する必要が出てくる可能性がある．

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異常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，不測の事態によって，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ある．

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 [5] 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[4] 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[4] とは，2.1.1 などを用いて取得した静的なメモリダンプに対して，解析を行うソフトウェアである．

Volatility では，取得したメモリダンプがアトミックである前提のもと，オペレーティングシステムがどのような状態にあったかを解析するためのものである．Volatility で使用されている手法は本研究において大いに参考になるが，このソフトウェアは，静的なファイルにのみ対応している．つまり，動作中のコンピュータに対する解析を行うことはできない．

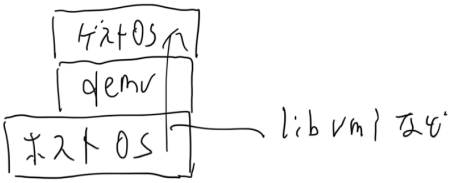
2.1.2 VM を用いた解析

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[3] 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[2] 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

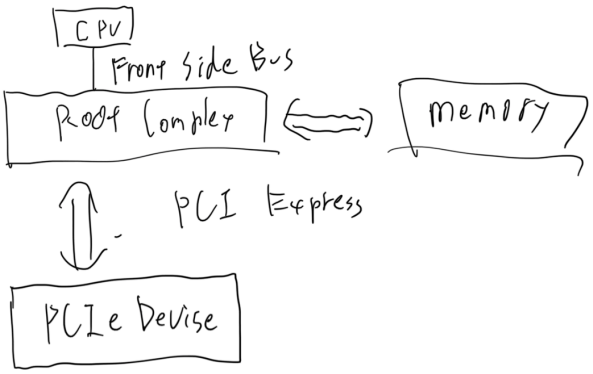
QEMU の上で実行する libvmi[9] というものがあり．これを使用した解析も可能である．

詳しく書く

2.2 RDMA

RDMA（Remote Direct Memory Address）とは，DMA（Direct Memory Access）[7] 転送をネットワーク越しに行う技術のことである．DMA は，メモリを読まれる対象のホストのマザーボード上の PCIeBus の上で動作する規格であるため，メモリを読まれる対象のホストの CPU コアを介さない通信が可能である．RDMA では，ネットワーク越しに DMA message を発行する技術であり，解析の際に CPU コアのリソースを使用しない，ゼロ・オーバーヘッド動作環境を実現することができる．さらに，監視対象ホストのオペレーティングシステムの状態に依存しない，つまり，電源さえ入っている状態であれば，動作中であろうとカーネルパニックが発生している状態であろうと，DMA message を発行し，結果を得ることが可能となる．

図 2.2: PCI Express



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

RDMA の実装として，Infiniband[8] における RDMA 実装がある．

しかし，infiniband における RDMA[1] では，実際のパケットの送受信を行うのは，Host Channel Adapter(HCA) であるが，この HCA がアクセスできる領域はあらかじめ Memory Region として監視対象ホストの OS にされている．アクセスできる領域の仮想アドレスと物理アドレスの変換表は HCA が保持し，変換した上でアクセスする．そのため，infiniband RDMA では規格上，許可された仮想アドレス空間の指定しかできず，メモリ空間の全てを参照することはできない．したがって，infiniband RDMA では，オペレーティングシステム全体の監視・解析を行うことは難しい．

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．これは，カーネルのコードを実行する際も同様である．

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．

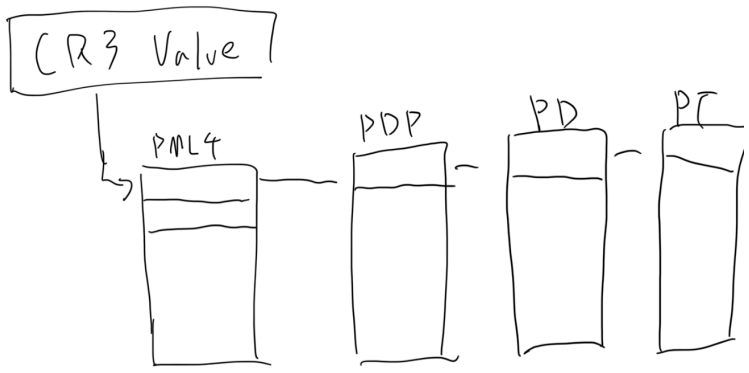
3.1.3 ホスト自身によるレジスタやシンボルの参照

3.1 で述べたように，オペレーティングシステムでは，その実行中のコンテキストにおいて，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．

さらにレジスタを参照して，Memory Management Unit を通じ，ページウォークなどをすることも可能となっている．

本研究では，メモリの情報のみ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することを試みるが，CPU レジスタ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，どのフィールド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，動作中のコンピュータのメモリを探索することで，ネットワーク越しにある物理的な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．そこで，本研究では，事前に解析者が保持する情報として，監視対象ホストのオペレーティングシステムの種類とバージョン情報を与えるものとする．

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など，オペレーティングシステムの状態を保持する領域は，メモリから知ることはできない．そのため，オペレーティングシステムの内部のシンボル，一例として，プロセス情報を保持するシンボルおよびその型情報を復元することを試みる．

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[6] 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

NetTLP[6] のユースケースの一つとして，process-list.c が実装されている．このプログラムでは，引数として監視対象ホストの/boot/config ファイルを受け取り，そのファイルの中身を検索している．すなわち，pid 0 を持つプロセスの情報として，監視対象ホストの init_task の情報を

定めている．与えられた init_task の開始アドレスから，連結リストとなっている task_struct を全て辿ることを試みている．

しかしこの実装は，task_struct の各フィールドのオフセットに関する値やマクロによって決定されるべき値がハードコーディングされているため，論文中の実行環境以外で実行することが困難となっている．この，task_struct の各フィールドのオフセットに関する値やマクロによって決定されるべき値は，Linux カーネルのバージョンと，カーネルコンフィグの値によって決まるが．本研究では，このプログラムを，ある特定のバージョンであれば，どのようなカーネルコンフィグを持っていても動作することができるように変更をする．具体的な変更内容については，4.10 にて述べる．

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したプログラム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．

実装したプログラムを実行するホストは，Linux 4.19.0-6-amd64 の Debian buster であり，LCLC ケーブルに対応した 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．

4.4 実装の前提情報

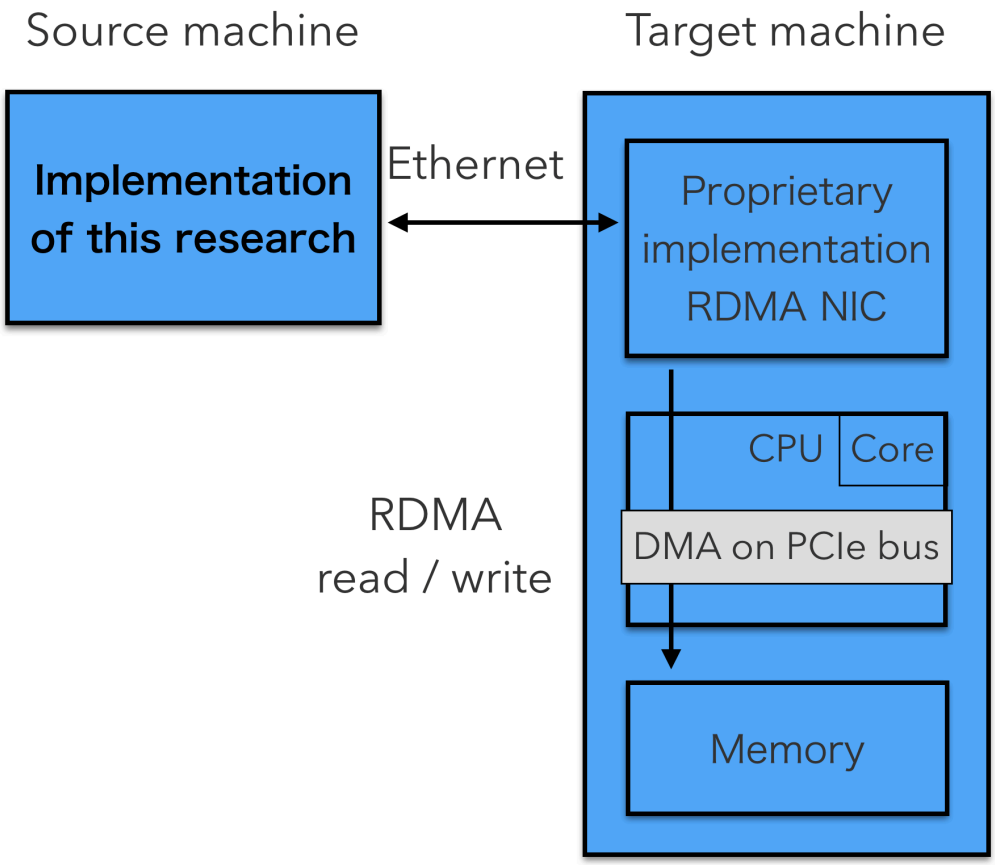
本研究では，3 章で述べたように，動作中のコンピュータのメインメモリの値を読むことによって監視対象ホストにおけるオペレーティングシステムのコンテキストを復元することを目的としている．その手法として，RDMA NIC を物理的に設置することで，この目的を達成することを試みている．

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，実装したプログラム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

図 4.1: 全体


Source machine
Target machine
Implementation of this research
Ethernet
Proprietary implementation RDMA NIC
CPU
Core
DMA on PCIe bus
RDMA read / write
Memory

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，実験に際して実装したプログラム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したプログラム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

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，実装したプログラム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

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，64bit 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したプログラム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: 実装したプログラムを実行するホスト

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，実装したプログラム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，実装したプログラム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したプログラム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．実行の際は，引数として，5.3.5 で導出した値を実行時に渡す．

process-list の出力

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

実行結果は，5.4.2 に添付してあるが，プロセス ID 0 を持つプロセスに始まり，全てのプロセスを取得できている．

比較対象として，監視対象ホストにて実行した ps コマンドの出力結果を用意した．実行結果は，5.4.2 に添付してあるが，この結果と先ほど 5.4.2 で取得した結果を比較してみると一致していることがわかる．

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．

この評価では，特定のプロセスに関する名前とプロセス ID に関する情報が正しく取得できているかを示す．手法として，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，そのプロセスに関する行を，検索し取得する．取得した情報のうち，プロセス ID が一致していることを示す．

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 --user
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 --exclude-dir=.bzr --exclude-dir=CVS --exclude-dir=.git
--exclude-dir=.hg --exclude-dir=.svn user
```

監視対象ホストで ps コマンドを実行した結果，user というコマンド名を持つプロセスの ID は 4189 であることがわかる．一方で，process-list の出力結果から user という文字列で検索をかけた結果以下のような行が抽出できた．

process-list から user というプロセスを検索

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

プロセス ID として 4189 を持つプロセスを取得できていることがわかる．

user という名前を持つプロセスのプロセス ID が一致していることが確認できたため，process-list.c が正しくプロセスの情報を取得できていることを示した

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．

本セクションでは，物理マシンがカーネルパニックを起こした際に，本研究の実装を実行し，コンピュータの最後の状態を取得できることを示す．

物理マシンに対して，監視対象ホストのプロセスとして起動する方式では，監視対象ホストでカーネルパニックが起きた際にプロセスそのものが停止してしまい，監視を続けることができなくなってしまう．しかし本研究で用いる NetTLP 環境は，電源さえ入っていれば監視対象ホスト

がカーネルパニックを起こした際にも動作可能であるため，カーネルパニック発生後にプロセス情報の一覧を取得することを試みる．

評価の準備として，以下のコマンドを監視対象ホストで実行し，意図的にカーネルパニックを引き起こす．

意図的にカーネルパニックを発生させる

```
sudo sh -c 'echo 1 > /proc/sys/kernel/sysrq'
sudo sh -c 'echo c > /proc/sysrq-trigger
```

コマンド実行後，process-list.c を実行した結果が以下である．最後に実行したコマンドは c という文字列を/proc/sysrq-trigger に出力するコマンドである．出力のうち，プロセス ID 1318 という行があるが，これが最後に正しく実行した命令である．カーネルパニック発生時にも，オペレーティングシステムが正常に動作していた時の情報を監視対象ホストから取得できることを示した．

1

```
./process-list 0xFFFFFFFF82413480
init_vm_addr: 0xffffffff82413480
PhyAddr             PID STAT COMMAND
0x00000002413480      0 R: swapper/0
0x000002361f16c0      1 S: systemd
0x0000022e35ad80    273 S: systemd-journal
0x0000022ea70000    295 S: blkmapd
0x0000022ea30000    298 S: systemd-udevd
0x0000022e35db00    562 S: rpcbind
0x000002325f5b00    565 S: systemd-timesyn
0x000002325f4440    567 S: atd
0x000002325f2d80    568 S: rsyslogd
0x000002325f0000    569 S: networkd-dispat
0x00000234be0000    571 S: irqbalance
0x0000022d858000    577 S: cron
0x0000022e6a5b00    580 S: systemd-logind
0x0000022ea20000    581 S: accounts-daemon
0x0000022ea25b00    582 S: dbus-daemon
0x0000022eaa96c0    598 S: wpa_supplicant
0x000002337c8000    642 S: polkitd
0x0000022db1c440    669 S: systemd-resolve
0x000002337cdb00    698 S: dhclient
0x00000231ccdb00    815 S: unattended-upgr
0x00000231ccad80    816 S: nmbd
0x00000231cc8000    817 S: sshd
0x0000022eaac440    843 S: sshd
0x00000233ab8000    954 S: sshd
0x0000022ea716c0    955 S: zsh
0x0000022eaa8000   1317 S: sudo
0x0000023357db00   1318 R: sh
0x00000231cc96c0    821 S: agetty
0x0000022eaaad80    836 S: smbd
0x000002325796c0    838 S: smbd-notifyd
0x0000023257ad80    839 S: cleanupd
0x00000232578000    840 S: lpqd
0x000002325f16c0    845 S: systemd
0x00000235d5db00    846 S: (sd-pam)
0x000002337c96c0   1299 S: certbot
```

2

```
0x000002361f0000      2 S: kthreadd
0x000002361f5b00      3 D: kworker/0:0
0x000002361f4440      4 D: kworker/0:0H
0x000002361f2d80      5 D: kworker/u8:0
0x0000023622db00      6 D: mm_percpu_wq
0x0000023622c440      7 S: ksoftirqd/0
0x0000023622ad80      8 D: rcu_sched
0x000002362296c0      9 D: rcu_bh
0x00000236228000     10 S: migration/0
0x00000236255b00     11 S: watchdog/0
0x00000236250000     12 S: cpuhp/0
0x0000023625c440     13 S: cpuhp/1
0x0000023625ad80     14 S: watchdog/1
0x000002362596c0     15 S: migration/1
0x00000236258000     16 S: ksoftirqd/1
0x0000023625db00     17 D: kworker/1:0
0x0000023630db00     18 D: kworker/1:0H
0x0000023630c440     19 S: cpuhp/2
0x0000023630ad80     20 S: watchdog/2
0x000002363096c0     21 S: migration/2
0x00000236308000     22 S: ksoftirqd/2
0x00000236372d80     23 D: kworker/2:0
0x000002363716c0     24 D: kworker/2:0H
0x00000236370000     25 S: cpuhp/3
0x00000236375b00     26 S: watchdog/3
0x00000236374440     27 S: migration/3
0x000002363dad80     28 S: ksoftirqd/3
0x000002363d96c0     29 D: kworker/3:0
0x000002363d8000     30 D: kworker/3:0H
0x00000235c52d80     31 S: kdevtmpfs
0x00000235c9db00     32 D: netns
0x00000235c9c440     33 S: rcu_tasks_kthre
0x00000235c9ad80     34 S: kauditd
0x00000235c996c0     35 D: kworker/0:1
0x00000235c98000     36 D: kworker/1:1
0x00000235d5ad80     37 S: khungtaskd
0x00000235d596c0     38 S: oom_reaper
0x00000235d68000     39 D: writeback
0x00000235d6db00     40 S:
```

3

```
0x00000235d6c440     41 S: ksmd
0x00000235d6ad80     42 S: khugepaged
0x000002363ddb00     43 D: crypto
0x000002363dc440     44 D: kintegrityd
0x00000235d7ad80     45 D: kblockd
0x00000235d796c0     46 D: kworker/2:1
0x00000235d78000     47 D: kworker/3:1
0x00000235d7db00     48 D: ata_sff
0x00000235d7c440     49 D: md
0x00000235eb2d80     50 D: edac-poller
0x00000235eb16c0     51 D: devfreq_wq
0x00000235eb0000     52 D: watchdogd
0x00000235c516c0     53 D: kworker/u8:1
0x00000235c55b00     55 S: kswapd0
0x00000235c54440     56 D: kworker/u9:0
0x0000022ea02d80     57 S: ecryptfs-kthrea
0x00000235eb5b00     99 D: kthrotld
0x00000235eb4440    100 D: acpi_thermal_pm
0x0000022ea00000    101 D: kworker/0:2
0x00000235d696c0    102 D: kworker/u8:2
0x00000235c50000    103 D: kworker/2:2
0x0000022ea74440    105 D: kworker/1:2
0x0000022ea04440    109 D: ipv6_addrconf
0x0000022ea10000    118 D: kstrp
0x0000022ea72d80    135 D: charger_manager
0x0000022e6a2d80    180 S: scsi_eh_0
0x0000022e6a4440    181 D: scsi_tmf_0
0x00000235d5c440    182 S: scsi_eh_1
0x0000022ea2ad80    183 D: scsi_tmf_1
0x0000022ea2db00    184 S: scsi_eh_2
0x0000022ea18000    185 D: scsi_tmf_2
0x0000022ea1db00    186 S: scsi_eh_3
0x0000022ea196c0    187 D: scsi_tmf_3
0x0000022ea28000    188 S: scsi_eh_4
0x0000022ea296c0    189 D: scsi_tmf_4
0x0000022ea15b00    190 S: scsi_eh_5
0x0000022ea1c440    191 D: scsi_tmf_5
0x0000022d852d80    192 D: kworker/u8:3
```

4

```
0x0000022d8516c0    193 D: kworker/u8:4
0x0000022d850000    194 D: kworker/u8:5
0x0000022d855b00    195 D: kworker/u8:6
0x0000022d854440    196 D: kworker/u8:7
0x0000022d85db00    197 D: e1000e
0x0000022d85c440    198 D: e1000e
0x0000022d85ad80    200 D: kworker/1:1H
0x0000022d8596c0    202 D: kworker/2:1H
0x0000022ea22d80    231 D: jbd2/sda1-8
0x0000022ea24440    232 D: ext4-rsv-conver
0x0000022e6a0000    234 D: kworker/0:1H
0x0000022e6a16c0    246 D: kworker/3:1H
0x0000022ea34440    275 D: kworker/3:2
0x00000234fc96c0    278 D: rpciod
0x00000234fc8000    279 D: xprtiod
0x00000234be2d80    370 D: ttm_swap
```

5.5 評価のまとめ

本研究の実装に対する評価として，オペレーティングシステムのコンテキストの復元を行う上で，監視対象ホストのカーネルコンフィグに依存する情報を正しく復元できていることを示した．

また，4 章で述べた構成の元，ネットワーク越しに存在している物理的なマシンのプロセス情報の一覧を正しく取得することで，RDMA を用いたメモリ探索を行うことで，オペレーティングシステムの復元をすることが可能であることを示した．

さらに，監視対象ホストでカーネルパニックが発生した際にも，オペレーティングシステムが正常に動いていた最後の状態を取得できていることを示した．

第6章　まとめと結論

6.1　まとめ

本論文のまとめとして，各章の内容を述べる．

1 章では、本研究の背景および課題として，大規模データセンタのコンピュータ管理者にとって，様々な設定をもつ大量の物理的なコンピュータの監視および解析が困難であることを述べたそこで本研究の目的として，

2 章では、仮想環境によるオペレーティングシステムのデバッグや，監視対象ホスト内で監視プロセスを起動する手法など，既存のオペレーティングシステムのコンテキストを監視する手法について述べた．さらに，様々な RDMA 実装がある中で，本研究で使用する NetTLP による RDMA を使う理由について述べた．

3 章では、動作中のコンピュータから取得したアトミックではないメモリダンプ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ための手法について述べた．その上で，メモリからどのような情報を探索することで，コンピュータの状態を復元できるのかについて述べた．さらに，物理メモリアドレスのみを指定できる中で，取得が困難な情報がどのような種類の情報で，その情報を本研究においてどのように復元していくかについて述べた．

4 章では、3 章で述べた手法を実現するための具体的な実装について述べた．特にメモリダンプしかない状態からいかにして，監視対象ホストのカーネルコンフィグの値を復元するか，復元した値からコンピュータの内部的な値，すなわち構造体のオフセットを復元する実装について述べた．最終的に，Linux カーネルのバージョンのみを通知された状態から，プロセス一覧に関する情報を取得するために必要な値を復元し，プロセス一覧を出力できる実装について述べた．

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．実験として，4 章で述べた工程を一つずつ実行した過程を示した．最終的に，本研究の実装の出力結果と監視対象ホストで実行した ps コマンドの出力結果を比較し，任意に起動したプロセスの ID が等しくなっていることを示した．さらに，監視対象ホストにおいてカーネルパニックを発生させ，オペレーティングシステムが停止した状態の中でも，プロセス情報の一覧を取得できることを示した．

6.2　結論

本研究の結論として，4 章で述べた実装を用いることで，監視対象ホストのバージョン情報のみを知らされた状態で，オペレーティングシステムのコンテキストの一つであるプロセス情報の

一覧を取得できることを示した．

プロセス一覧を探索するにあたり，監視対象ホストが内部で使用している値，例えば，task_struct 構造体の各フィールドのオフセットや，init_task のカーネル空間における仮想アドレスを，自ホストで推定，導出するために，メモリダンプから収集したカーネルコンフィグの値から，実装したプログラムを実行するホストで再度ビルドすることで，復元できることを示した．

また，カーネル空間はストレートマップであるがゆえに，メモリダンプから特定の値，本研究の実験では，swapper/0 という文字列を走査し，そこから物理アドレスおよびカーネル空間における仮想アドレスを導出することができることを示した．

6.3 今後の課題

本研究の実験における環境として，オペレーティングシステムの情報，すなわち Linux カーネルのバージョンに関する情報は，事前に実装したプログラムを実行するホストは知っていることとした．しかし，現実のコンピュータは特定の Linux カーネルのバージョンで動いているわけではない．特に大規模データセンタにおいては，各ホストは様々なカーネルバージョンおよびディストリビュージョンで動作している．

よって今後の課題として，メモリダンプの情報から Linux カーネルのバージョンを特定することをあげる．

6.3.1 セキュリティ的な課題

本研究における環境では，FPGA ボードを物理的に設置する，という工程のみでメモリの情報を取得でき，オペレーティングシステムのコンテキストを復元できてしまう．その一例として，本研究の実装では，プロセス情報の一覧を，Linux カーネルのバージョン情報のみから復元できることを示した．

当然，ルート権限はおろか，通常のユーザー権限すらない中での復元となるため，悪用された場合に存在そのものがセキュリティ的なリスクとなってしまう．そのため，今後の研究では，6.3 で述べた課題に加えて，本研究のセキュリティ面におけるリスク軽減に関する研究を行っていく．

謝辞

アドバイスをくれた全員に感謝

参考文献

[1] . http://www.nminoru.jp/~nminoru/network/infiniband/iba-concept.html.

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

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

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

[5] 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.

[6] 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.

[7] Nader Amini, Patrick M Bland, Bechara F Boury, Richard G Hofmann, and Terence J Lohman. System direct memory access (dma) support logic for pci based computer system, September 12 1995. US Patent 5,450,551.

[8] Nusrat S Islam, Mohammad Wahidur Rahman, Jithin Jose, Raghunath Rajachandrasekar, Hao Wang, Hari Subramoni, Chet Murthy, and Dhabaleswar K Panda. High performance rdma-based design of hdfs over infiniband. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–12. IEEE, 2012.

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