メモリ解析におけるグラフを用いた物理アドレスから仮想アドレスへの逆変換手法

大月 勇人[1,a)]　碓井 利宣[1]　塩治 榮太朗[1]　川古谷 裕平[1]　岩村 誠[1]

概要：物理アドレスから仮想アドレスへの変換（逆変換）はメモリフォレンジックス，マルウェア解析，脆弱性検出などの，特に低レイヤを対象とする解析において，必須の技術である．しかし，現在のコンピュータシステムではこの逆変換は想定されておらず，現実的な計算量と精度では実現できない．そこで，本論文では，ページテーブルが木構造に類似した構造をしていることに着目し，テーブルを有向グラフに変換することで，高速かつ正確な逆変換を実現する手法を提案する．既存手法との比較実験では，正確性重視の手法と比べて，同等の正確性を 40 倍高速かつ 900 分の 1 のメモリ使用量で実現できた．さらに，高速性重視の手法が 1%程度の変換精度なのに対し，提案手法は現実的な実行時間で精度 100%を達成した．

キーワード：メモリ解析，フォレンジックス，リバースエンジニアリング，ページテーブル，有向グラフ

Graph-based Physical-to-Virtual Address Translation for Memory Analysis

Yuto Otsuki[1,a)]　Toshinori Usui[1]　Eitaro Shioji[1]　Yuhei Kawakoya[1]　Makoto Iwamura[1]

Abstract: Physical-to-virtual address translation (P2V translation) is an essential technique for malware analysis, forensics, vulnerability assessment, and so on. However, existing techniques are computationally expensive or inaccurate since modern computer system is not designed for P2V translation. In this paper, we propose a technique for lightweight and accurate P2V translation, focusing on the structure of a page table which is similar to tree structure. The proposed technique achieves practical P2V translation by converting page tables to a directed graph and using path-finding techniques. We conducted an experiment and show that our technique is 40x faster and consumes 900x less memory than existing accuracy-oriented techniques. We also show that our technique has a translation accuracy of 100% within practical time, whereas existing speed-oriented techniques have an accuracy of about 1%.

Keywords: Memory Analysis, Forensics, Reverse Engineering, Page Table, Directed Graph

1. はじめに

マルウェア解析やフォレンジックス，脆弱性検証などでは，プログラムやシステムの状態や挙動の解析が不可欠である．そのために用いられる手段の 1 つとしてメモリ解析がある．メモリ解析は，一般に，解析対象プログラムを実行している仮想メモリ空間を対象として行われる．これは，解析対象と同じ視点でメモリ上のデータを参照することにより，その状況の分析や情報の収集を行うためである．

しかし，システムワイドの解析が必要な場合では，仮想メモリ空間を対象とすると計算量が膨大になる．例えば，共有メモリ領域の検出やマルウェアのコードインジェクションの解析を行う場合，全てのプロセスの仮想メモリ空間から解析を行おうとすると，対象範囲が広大[*1]になり，現実的に困難な計算量になってしまう．

[1] NTT 社会情報研究所
NTT Social Informatics Laboratories
[a)] yut.otsuki@ntt.com

[*1] 例えば，x64 アーキテクチャ，4 階層ページングの場合，ページの数は最大で「$2^{(48-12)}$ × プロセス数」となる．

これを解決するために，物理メモリを起点に解析を行う方法が考えられる．そのためには物理アドレスから対応する仮想アドレスを取得（逆変換）できる必要がある．現在のコンピュータアーキテクチャやオペレーティングシステム (OS) では，仮想アドレスから物理アドレスへの変換 (Virtual-to-Physical 変換，V2P 変換) のみが想定されており，逆変換 (Physical-to-Virtual 変換，P2V 変換) は想定されていない．そのため，その実現は容易ではない．

P2V 変換の既存研究はいくつか存在し，V2P 変換の結果に基づくもの (`pas2vas`[1], `phys2virt`[2], `p2v`[3]) と，OS のメモリ管理データを参照するもの (`ptov`[1], `P2V`[4]) に大別される．前者は，正確かつ網羅的な変換が可能であるが，これ自体が仮想メモリ空間からの解析であるため計算量に課題がある．後者は高速ではあるが，P2V 変換の可否が OS 側の設計に依存し，正確性や網羅性に課題がある．

そこで，本論文では，正確性と高速性を両立した，OS 非依存の P2V 変換手法を提案する．提案手法では，ページテーブルが木構造に類似した構造をしていることに着目し，テーブルを有向グラフに変換する．これにより，アドレスの変換も経路探索の問題に変換して解くことを可能にしている．また，グラフの構造に基づいて逆変換結果を予測し，連続する仮想アドレスを範囲表現にまとめることで，結果の列挙にかかる計算量も削減している．

実験では，提案手法と既存実装をメモリフォレンジックスツール Rekall [5] に実装し，比較を行った．具体的には，解析対象の物理メモリダンプに含まれる全ての物理ページのアドレスに対して P2V 変換を行い，正確性とパフォーマンスの評価を確認した．これにより，提案手法は，正確性重視の手法と同等の正確性でありながら，それと比べて 40 倍高速であり，かつメモリ使用量は 900 分の 1 で実行可能であることを確認した．また，高速性重視の手法が正解データと比べて 1%程度しか一致しなかったのに対し，提案手法は 100%一致を現実的な実行時間で達成した．

本論文の貢献は以下の 3 点である．

- 既存の P2V 変換手法を 2 種類に分類し，一方は正確であるが計算量が現実的でなく，もう一方は高速であるが正確性が低いという課題があることを示した．
- ページテーブルをグラフに変換することにより P2V 変換を実現する手法を新たに提案した．
- 提案手法が，正確性重視の既存手法と比べて，40 倍高速かつ 900 分の 1 のメモリ使用量で P2V 変換が可能であることを実験で示した．さらに，高速性重視の手法が 1%程度の変換精度なのに対し，提案手法は実用的な速度で 100%の精度を達成した．

2. 背景

本章では，まず，x64 アーキテクチャにおける 4 階層のページテーブルについて説明する．さらに，P2V 変換が有効な利用ケースを整理し，その必要性を示す．

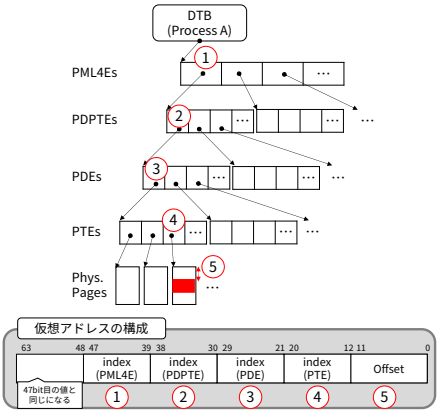

図 1　4 階層ページテーブルと仮想アドレス (x64)
Fig. 1　Page table and virtual address for 4-level paging (x64).

2.1 x64 における 4 階層ページング

図 1 は，x64 アーキテクチャにおける 4 階層のページテーブルと仮想アドレスの例を示している．ページテーブルは多段階のテーブルで構成されており，各テーブルのエントリには次の階層の物理アドレスが含まれている．仮想アドレスは，各階層のテーブルを参照するためのインデックスと，ページ内のオフセットから構成される．テーブルエントリは階層ごとに呼び名が変わり，上から PML4E, PDPTE, PDE, PTE と呼称される．

ただし，Large Page 等と呼ばれる機能により，ページテーブルの階層が少なくなる場合がある．x64 アーキテクチャでは，ページサイズは原則 4KB であるが，一部に例外的に 2MB もしくは 1GB 分の仮想ページを同じサイズの連続する物理ページにまとめて対応付けることが可能となっている．例えば，2MB の Large Page の場合，PDE から直接 2MB 分の連続した物理ページが示されるようになる．

なお，OS は原則プロセスごとにページテーブルを構築することで，それぞれに固有の仮想メモリ空間を提供する．このため，最上段のテーブルの物理アドレスがプロセスごとに固有となる．本論文では，当該テーブルのベースアドレスを DTB (Directory Table Base) と呼称する．

2.2 P2V 変換が有用なケース

本節では，P2V 変換が有効な 2 つの利用ケースを挙げ，特にマルウェア解析や脆弱性検証の観点から説明する．

2.2.1 共有メモリ領域の検出

共有メモリ領域は，1 つ物理ページを複数の仮想ページにマップすることで，プロセス間やユーザ・カーネル間でのデータの共有を可能にする．マルウェアのコードインジェクション（Section Mapping Injection [6], PowerLoaderEx [7], malWASH [8] など）や脆弱性攻撃（win32k への攻撃 [9]

など）にも利用されることがあるため，共有メモリ領域を特定し，その内容の解析や，その領域を利用しているプロセスの列挙が必要となる．

上記のようなケースでは，共有メモリ領域の検出が解析に不可欠である．しかし，共有メモリ領域は，仮想メモリ空間からは独立して存在しているように見えるため，メモリ解析での検出は容易ではない．従来は，実現に OS のメモリ管理用のデータ構造を解析する必要があったが，例えば Windows ではドキュメント化されていない部分も多く，更新による細かな仕様変更もありうるため，常に安定して使えるとは限らない．もし実用的な P2V 変換が実現できれば，共有メモリとして使用されているページの検出が容易になる．具体的には，ある物理ページのアドレスを P2V 変換し，対応する仮想アドレスが複数取得できる場合，そのページが複数の仮想ページにマップされていることを示している．特に，異なるプロセスに属する仮想アドレスなど，異なる空間の仮想アドレスが複数取得できれば，それらを共有メモリ領域として検出できる．

2.2.2 物理アドレスを起点にしたメモリスキャン

メモリスキャンは，シグネチャを用いてメモリ空間をスキャンし，特定のデータの位置を特定する手法である．マルウェアや攻撃コードの検出に広く使用されている．しかし，現在の仮想メモリ空間は広大である上に，プロセスの数分存在するため，全域をスキャンするのは現実的でない．

これに対し，実体である物理メモリへのスキャンと P2V 変換を組み合わせることで，効率化が可能である [1]．具体的には，物理メモリ空間へのスキャンで検出したデータの物理アドレスに対し，P2V 変換を行うことで，使用しているプロセスやその仮想アドレスを特定することができる．これにより，仮想メモリ空間でのスキャンと同等の情報を提供することが可能となる．

3. 既存手法と課題

本章では，既存の P2V 変換手法について述べ，その課題を整理する．文献 [1] では，Rekall [5] のプラグインの `pas2vas` と `ptov` について言及されている．当該文献以降に公開された手法や実装も調査したところ，どれも上記の 2 つと類似した方式で実現されていた．本論文では，既存の P2V 変換手法について，全列挙方式と，OS データ構造参照方式の 2 種類に分類し，`pas2vas` と `ptov` を代表例として説明する．なお，他の例については 6 章で述べる．

3.1 全列挙方式

全列挙方式は，実際にページテーブルを一つずつ探索し，仮想アドレスと物理アドレスの組み合わせを列挙することで実現している．この方式は V2P 変換の結果に基づくため，正確かつ網羅的な逆変換が可能である．

既存の実装例として，Rekall の `pas2vas` プラグインが挙げられる．`pas2vas` では，列挙した結果を用いて逆変換用のテーブルを構築し，それに基づいて逆変換を行う．しかし，現在の仮想メモリ空間の大きさでは組み合わせの数が膨大となるため，現実的な計算時間及びメモリ使用量ではテーブルの構築が終わらない場合がある．また，カーネル空間は共通であることを前提としており，OS が持つ DTB 以外でのカーネル空間分のテーブル作成を省略する．実際には，例えば Windows の場合，カーネル空間内でも共通でないエリアが存在する [10], [11]．そのため，これでは逆変換が不完全になってしまう．

3.2 OS データ構造参照方式

別の方式として，OS が内部的に構築している独自のメモリ管理用のデータ構造を解析して逆変換を実現する方法が挙げられる．OS 独自の管理用のデータ構造に依存した方法であるため，OS ごとに個別に P2V 変換を設計することが必要となる．また，OS が P2V 変換を想定してデータ構造を設計しているとは限らない．そのため，必要な情報が十分に含まれておらず，正しく変換できない場合もある．

Rekall の `ptov` は，Windows を対象とするこの方式の実装である．`ptov` は，前述の `pas2vas` と比べて高速に動作する反面，正しく逆変換可能な物理アドレスが限定的である．具体的には，当該プラグインでは，物理ページを管理するためのデータ構造である MmPfnDatabase を参照することで逆変換を実現する．ただし，これのみでは，共有ページを逆変換できないため，各プロセスのユーザ空間のレイアウトを管理するデータ構造 VAD (Virtual Address Descriptor) ツリー [12] の解析も行う．これにより，ユーザ空間内の共有領域について補完可能にしている．

しかし，それでもカーネル空間内の共有ページや，カーネルとユーザ間の共有ページには対応できないため完全な対策にはなっていない．また，当該プラグインでは Large Page 内の物理アドレスも正しく逆変換できない．

4. 提案手法

ここまでで P2V 変換が必要なケースが存在しているが，既存手法には課題も多く実用的ではないことを述べた．本章では，それらを解決した新たな P2V 変換手法を提案する．

4.1 概要

提案手法では，ページテーブルの構造が木構造に近いことに着目し，グラフに変換することにより，P2V 変換を可能にしている．具体的には，ページテーブルの構造を解析し，テーブルを構成する物理ページをノードとし，各エントリが示す物理ページ間のリンクをエッジとする，多重辺有向グラフを構築する．グラフ化により，アドレス変換を

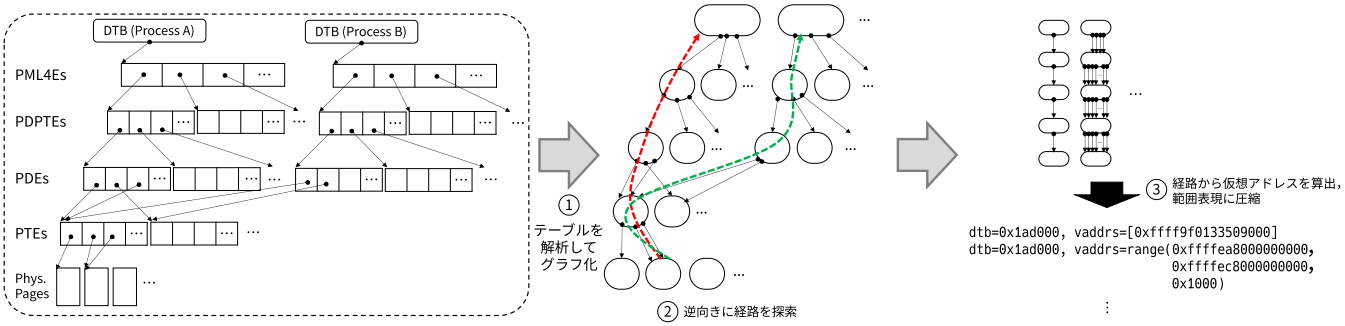

図 2　提案手法の処理の流れ
Fig. 2　Workflow of the proposed technique.

経路探索問題として解くことを可能にする．P2V 変換は，変換したい物理アドレスを含む物理ページに対応するノードから DTB に対応するノードまでの経路を探索し，得られた経路から仮想アドレスを算出することで実現できる．

仮想ページの数は，各階層のテーブルインデックスの組み合わせ数であるため膨大な数となる．3.1 節で述べた全列挙方式では，事前にすべての仮想ページの完全なアドレスを算出する必要があるため，必要な計算量も大きくなっていた．一方，仮想ページの数と比べるとテーブルの構成要素そのものの数は小さく，テーブルの一部は共通されることも多い．そのため，全列挙方式と比べて，グラフ化に要する計算量は抑えられる．さらに，提案手法では，1 つの物理アドレスに対して多数の仮想アドレスが得られる場合，連続する仮想アドレスを範囲表現にまとめることで結果を圧縮する．これにより，個々の仮想アドレスの列挙に要する計算量も削減している．

図 2 に提案手法の流れを示す．提案手法は，①グラフ構築，②経路探索，③仮想アドレスの算出・圧縮の 3 段階で構成される．以降，本章ではそれぞれの詳細を述べる．

4.2　①グラフ構築

提案手法は，始めに物理メモリデータと 1 つ以上の DTB を入力として受け取り，ページテーブルを解析し，グラフへの変換を行う．具体的なグラフ構築の流れを下記に示す．

(1) 受け取った全ての DTB が示す物理ページをノードとしてグラフに追加し，DTB としてマークする．
(2) (1) や (3) で新規追加されたノードに対応する物理ページを，ページテーブルの階層を構成するテーブルとして解析し，そのエントリを列挙する．
(3) 各エントリが示す次の物理ページに対応するノードがない場合，ノードとしてグラフに追加する．
(4) テーブルのノードから，各エントリが示す次の物理ページに対応するノードにエッジを張り，エッジには対応するエントリの情報を付加する．
(5) 全てのページテーブルの一番下の段まで探索が完了するまで (2) に戻り処理を繰り返す．

なお，提案手法では，上記 (3) や (4) において，エントリが示す先が Large Page として設定されていることを検出した場合は例外的な処理を行う．具体的には，そのエントリに対応するエッジに Large Page であることを示す情報を付加し，Large Page を構成する複数のページに対応するノードをまとめてグラフに追加する．

4.3　②経路探索

次に，提案手法は，構築したグラフに対し，変換したい物理アドレスに対応するノードから DTB ノードまでの経路探索を行う．なお，この時点では，ノード間に複数のエッジがある場合，どのエッジを進むかは考慮せずに経路を探索する．以降，区別のため，ノードの組み合わせのみによって示される経路をノード経路と呼び，ノード間でどのエッジを通るかを考慮した経路をエッジ経路と呼称する．

具体的な処理の流れを下記に示す．

(1) 逆変換したい物理アドレスを含むページに対応するノードを特定する．
(2) そのノードを起点としてエッジを逆向きに進み，ページテーブルの階層分のステップ数で到達するノードと，対応するノード経路を取得する．
(3) 最終到達点が DTB ではない経路を除外する．

なお，経路中に Large Page が含まれる場合，ここでも例外として取り扱う．ノード経路では，原則ノード間のどのエッジを通るかによらず 1 つの経路として扱うが，Large Page を含む経路については他の経路と区別する．

4.4　②仮想アドレスの算出・圧縮

最後に，提案手法は，前節で得られたノード経路を元に仮想アドレスを算出し，可能な範囲で範囲表現に変換することで，結果の圧縮を行う．

4.4.1　仮想アドレスの算出

まず，エッジ経路から仮想アドレスを算出する方法について説明する．具体的な流れを下記に示す．

(1) 仮想アドレスに変換したいエッジ経路に対し，ノード間を接続する各エッジに対応するエントリのテーブル

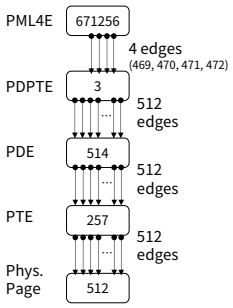

図 3 物理アドレス 0x200000 に対するサブグラフの一例
Fig. 3 A subgraph for physical address 0x200000.

内インデックスを取得する．

(2) 各インデックスと，物理アドレスに含まれるページ内オフセットを合成し，仮想アドレスに変換する．

(3) 経路の到達点である DTB からどの仮想メモリ空間に属する仮想アドレスかを判別する．

4.4.2 仮想アドレスの圧縮

ノード経路から対応するすべてのエッジ経路を取得し，前項で述べた方法を実施することで，物理アドレスに対応する全ての仮想アドレスの算出が可能である．しかし，1 つの物理アドレスに対して多数の仮想アドレスが対応づく場合，個別に算出するとやはり計算量が増大する．そこで提案手法では，仮想アドレスを変換するエッジ経路を限定する．具体的には，連続する仮想ページが 1 つの物理ページに対応づけられているケースをグラフの構造から検出し，その仮想アドレスを範囲表現に集約することで結果の出力に要する計算量を削減する．

提案手法では，ノード経路内でノード間が連続するエッジで接続されているものを含む場合に着目する．具体的には，ノード経路の開始点から DTB ノードに向かって，連続するエッジを検出していくことで，対応する仮想アドレスが連続することを判別する．

図 3 は，5 章で述べる実験で実際に観測された実例であり，物理アドレス `0x200000` に対する逆変換の過程で得られた経路の 1 つである．以降，この例を用いて仮想アドレスを範囲表現に圧縮する流れを説明する．物理アドレス `0x200000` に対応するノードは図中のノード 512 であるので，ここが経路の開始点である．ノード 512 からノード 3 からまでは，各ノード間で全てのエッジが同じノードに接続されている．すなわち，このサブグラフに対応する仮想アドレスは，12 ビット目から 38 ビット目までは任意の値を取り得る．ノード 671256 からノード 3 の間を接続する 4 つのエッジにおけるインデックスの最小値は 469，最大値は 472 であるため，算出される仮想アドレスの最小値は `0xffffea8000000000`，最大値は `0xffffec7ffffff000` となる．すなわち，この範囲に含まれる 536,870,912 種類の仮想ページが `0x200000` の物理ページに対応づけられていることがわかる．このように，提案手法では，経路の状態から連続する仮想ページが 1 つの物理ページに紐づくことを検出し，結果を範囲に集約することができる．

表 1 実験環境
Table 1 Experimental environment.

CPU	Intel(R) Core(TM) i7-8559U 2.70GHz (4 コア)
メモリ	10GB
OS (VM)	Ubuntu 22.04 LTS
Python	3.13.2 (free-threaded)

表 2 解析対象のメモリダンプ
Table 2 Target memory dump.

メモリサイズ	5GB (使用可能: 4GB)
OS	Windows 10 x64 2004 (Build 19041)
プロセス数	130
仮想メモリ空間数	203 (KDTB: 130, UDTB: 73)

5. 実験

5.1 実験概要

提案手法の基本性能を確認するために，メモリダンプの解析を行う実験を行った．本章では，逆変換結果の正確性とパフォーマンスを他の手法と比較した結果について述べる．また，実験中に確認した個別の逆変換結果について紹介し，提案手法の有効性を示す．

5.2 実験環境と解析対象

本実験の解析は，**表 1** に示す仮想計算機 (VM) 上で行った．解析対象として，Windows 10 x64 2004 を実行していた VM の物理メモリダンプを用いた．詳細を**表 2** に示す．メモリ解析は，Rekall を用いて行った．なお，実験で使用した Rekall には，今回解析対象としたバージョンの Windows を解析可能にするための改良を実施している．

解析対象のメモリダンプには，プロセスリストを解析するプラグインである `pslist` で取得可能なプロセスが 130 存在した[*2]．また，当該メモリダンプでは，Meltdown 対策である KVAS (Kernel Virtual Address Shadow) [13] が有効な環境であり，130 のプロセスのうち，73 で 2 種類のページテーブルを保有していた．そのため，仮想メモリ空間の数は 203 となる．以降，本論文では，ユーザ空間側で使用されるページテーブルの DTB を UDTB，カーネル側で利用される方の DTB を KDTB と呼称する．

5.3 比較用の実装

本実験では，**Simple**, **PageGraph**, **PFNDB**, **Rekall-Full** の 4 種類の P2V 変換手法を実装し，比較を行った．これらは全て Rekall のプラグインとして実装している．**Simple** は全列挙方式を実装したものであり，**PageGraph**

[*2] ダンプ取得時点で既に終了していたものも 8 つ含まれていたが，仮想メモリ空間がまだ有効であったため，解析対象に含めている．

表 3　正確性評価の結果
Table 3　Result of accuracy evaluation.

	総行数	一致行数	
Simple	39,153,476	—	
PageGraph	39,153,476	39,153,476	(100.00%)
PFNDB	630,248	282,983	(0.72%)
RekallFull	4,156,618	565,998	(1.45%)

が提案手法を実装にしたものである．**PFNDB** と **RekallFull** は OS データ構造参照方式の実装である．なお, Rekall には Windows 独自のページテーブル拡張を解析可能な機能が組み込まれており，例えば，未割り当てな仮想ページに割り当て予定の物理ページが取得可能な場合がある．本実験では，条件を揃えるために原則この機能は使用しない．

Simple は，Rekall の `pas2vas` を評価用に再現実装したものである．3.1 節で述べた通り，オリジナルの `pas2vas` には，精度を引き換えとした高速化の工夫が含まれているが，**Simple** ではそれは適用していない．また，完全な逆変換テーブルには少なくとも 1.54TB のメモリが必要と推定された[*3]ため，追加で下記の変更を行っている．

- 逆変換用のテーブル生成と逆変換の処理を分離し，個別に実行できるように変更した．
- 逆変換テーブルの実装，および結果の出力に提案手法の一部を適用し，適宜仮想アドレスを範囲表現に圧縮することで，テーブルのデータ量を削減させた．
- 逆変換テーブルは 1 つの仮想メモリ空間ごとに生成し，生成後にファイルにダンプした上で gzip 圧縮をかけるように変更した．逆変換時はダンプしたテーブルを個別に 1 つずつ読み込んで変換を行う．

PFNDB と **RekallFull** はどちらも Rekall の `ptov` プラグインを元にしている．**PFNDB** は, `ptov` の機能のうち，MmPfnDatabase に基づく逆変換のみを行う．一方，**RekallFull** は，`ptov` の全て機能を有効にしたものである．すなわち，MmPfnDatabase による変換だけでなく，VAD ツリーの解析結果に基づく補完機能も使用する．そのため，4 つのうち **RekallFull** のみ，結果に割り当て見込みの仮想アドレスも含まれる仕様となっている．

5.4　正確性評価

提案手法の正確さを確認するために，5.3 節で述べた 4 種類のプラグインで P2V 変換を行い，比較を行った．具体的には，各プラグインに対し，全ての物理ページのアドレスを与えて逆変換を実行し，結果をテキストファイルに出力させた．物理ページの総数は，使用不可エリアも含めて 1,310,720 である．本実験では，実際の V2P 変換結果に基づく **Simple** の結果を正解データとして評価を行う．

[*3] 解析対象のメモリダンプにおいて有効な仮想ページの総数は 105,502,326,715 であったため，その仮想アドレスを単に保持するだけでも 786GB は必要となる．

表 4　パフォーマンス評価の結果
Table 4　Result of performance evaluation.

	実行時間 (hh:mm:ss)		メモリ使用量
Simple	(テーブル生成)	推定 17 日	—
	(逆変換)	11:49:39	4.79GB
PageGraph		10:06:57	1.75GB

それぞれのプラグインから出力された結果の総行数と，**Simple** の結果と比較して一致した行数を**表 3** に示す．既存手法である **PFNDB** と **RekallFull** については，そもそも出力された結果の数が **Simple** と比べて大幅に少なかった．**PFNDB** で得られた結果の中の半数以上で，逆変換元の物理アドレスと実際には対応関係のない仮想アドレスが出力されていた．**RekallFull** では，VAD ツリーからの補完により **PFNDB** と比べて一致数は 2 倍に増えているが，総行数は 6.6 倍であることを考えると効果は限定的である．これは，**RekallFull** では，前述の通り，割り当て予定の物理ページも変換結果に含まれるためである．

一方，提案手法の実装である **PageGraph** は，**Simple** の結果と完全に一致しており，変換誤りや漏れがないことが確認できた．この結果より，提案手法は，既存手法と比べて正確な P2V 変換が可能であることが示せた．

5.5　パフォーマンス評価

本節では，提案手法の計算量を比較した結果について述べる．具体的には，5.4 節で実施した評価において，実行時間と最大メモリ使用量を `time` コマンドを用いて計測した．なお，**PFNDB** と **RekallFull** については，そもそも結果の出力数が正解の 1%前後しかないため，本比較から除外した．正確な P2V 変換が可能なことが確認できた **PageGraph** が，**Simple** と比べて高速であることを示す．

結果を**表 4** に示す．ただし，**Simple** ついては，前述の改良を加えた状態でも今回の実験環境では一度に実行できなかった．逆変換テーブル生成については，1 つのページテーブルに要した時間から算出した推定値を参考として示す．なお，17 日は並列処理をしなかった場合の推定値である．逆変換テーブルが生成済みの状態から計測したところ，変換完了まで約 12 時間を要した．一方，**PageGraph** は，グラフ構築から逆変換結果出力まで含めて 10 時間ほどで完遂している．グラフ構築のみを別途計測したところ約 20 秒であったため，**PageGraph** の処理時間の大半は逆変換に要した時間である．以上から，**Simple** と比較して，**PageGraph** は 40 倍高速であると言える[*4]．

メモリ使用量においても，**Simple** は逆変換処理のみで 4.79GB なのに対し，**PageGraph** は 1.75GB と 3 分の 1 程度となっている．本来は逆変換テーブルに推定 1.54TB

[*4] 仮に，**Simple** のテーブル生成を実験環境の CPU コア数分，すなわち 4 並列で実施したとしても 4 日間以上を要するため，その場合でも **PageGraph** の方が 11 倍高速である．

が必要であったことを考えると，提案手法は 900 分の 1 程度のメモリ消費量で逆変換を実現できたと言える．以上より，**PageGraph** が **Simple** と比べて高速かつ低メモリ消費量で P2V 変換が可能であることが確認できた．

5.6 事例検証

本節では，実験中に確認できた逆変換の事例 3 つについて述べる．具体的には，Rekall の **ptov** で課題とされていたものを提案手法で正しく変換できることや，変換結果出力における仮想アドレスの圧縮の有効性について言及する．また本節では，各事例について，1 つの物理アドレスを逆変換する場合にかかった実行時間も示す．本節で述べる **Simple** の実行時間については，原則逆変換テーブル構築済みの状態から逆変換にかかった時間を示す．

5.6.1 実例: Large Page

3.2 節で述べた通り，Large Page は既存の Rekall の `ptov` では正しく変換できない例の 1 つである．本実験で用いたメモリダンプでは，カーネルが Large Page にロードされていたため，これを検証に用いる．Rekall の解析により，仮想アドレスは `0xfffff80235e00000` であり，物理アドレスは `0x2400000` であることが確認できている．

この物理アドレス `0x2400000` に対して逆変換を行ったところ，**PFNDB** と **RekallFull** では，System プロセスの KDTB と `0xffffd2fc01000000` の組み合わせのみが得られた．しかし，当該仮想アドレスを再変換したところ，元の物理アドレスは得られなかった．

一方，**PageGraph** と **Simple** では，仮想アドレスと DTB の組み合わせが 260 件得られた．具体的には，全てのプロセスの KDTB 130 個について，それぞれ 2 種類の仮想アドレス `0xfffff80235e00000`，`0xffffd2fc011af000` が出力されていた．後者も再変換で正しく `0x2400000` が得られることを確認している．なお，逆変換に要した時間は **PageGraph** が 23 秒，**Simple** が 1 分半程度であった．以上から，OS データ構造参照方式の実装で対応できていなかった Large Page 内のページについても，提案手法で正しく逆変換が可能であることが確認できた．

5.6.2 実例: ユーザ・カーネル間の共有領域

ユーザ空間とカーネル空間で共有されているメモリ領域も，既存の `ptov` では正しく変換できない例の 1 つである．ここでは，その例として KUSER_SHARED_DATA [14] を用いて検証を行った．今回の解析対象では，原則全プロセス共通でユーザ空間では `0x7ffe0000`，カーネル空間では `0xfffff78000000000` のアドレスに配置されている．V2P 変換を行うと，どちらも `0x512d000` を示した．

当該物理アドレスに対して，各プラグインで P2V 変換を行った．**PFNDB**，**RekallFull** では変換ができず，結果は 0 件であった．一方，**PageGraph** と **Simple** では仮想アドレスと DTB の組み合わせが 318 件得られた．仮想アドレスとしては，`0x7ffe0000`，`0xfffff78000000000` の 2 種類であり，前者が 188 件[*5]，後者が 130 件である．なお，実行時間は前項の結果と同程度であった．以上から，ユーザ・カーネル間で共有されているページについても，提案手法で逆変換可能なことが確認できた．

5.6.3 実例: 極端な多重マッピング

本項では，4.4.2 項でも紹介した物理アドレス `0x200000` を例に提案手法の仮想アドレス圧縮の効果について示す．対象の物理アドレスには，図 3 で示した部分を含め，トータルで 105,238,829,540 通りの仮想アドレスが存在する．**PageGraph** は，最終的に 13,650 通りまで圧縮に成功し，P2V 変換に要した時間は約 39 秒であった．**Simple** でも同じ結果が得られており，かかった時間は 1 分 41 秒であった．なお，**PFNDB**，**RekallFull** は，どちらも仮想アドレスが 1 つ得られたのみで，その値も間違っていた．以上より，提案手法は，対応する仮想アドレスの数が膨大な場合でも，現実的な計算量で結果を得られることが確認できた．

6. 関連研究

本章では，既存の P2V 変換手法について述べる．なお，文献 [1]，および Rekall の `pas2vas`, `ptov` については 3 章で述べたため，本章ではその他の既存手法について述べる．

物理メモリ解析ツール MemProcFS [2] でも P2V 変換を行う `phys2virt` という機能が提供されている．この機能は，全列挙方式に該当する実装であるが，事前に逆変換用のテーブルは作成せず，都度ページテーブルを直接参照して逆変換を行う．また，`phys2virt` では，逆変換結果を各プロセスごとに最大 4 つに制限している．これにより，メモリ消費量と処理時間を抑えていると考えられる．提案手法は網羅的な P2V 変換を実現することを目的としているため，この `phys2virt` とは設計意図が異なる．

デバッガでの実装例として，GDB 向け拡張スクリプト gef の fork [3] において，QEMU [15] を用いたデバッグ用途で使用できる `p2v` コマンドが提供されている．これも全列挙方式に分類でき，`pas2vas` と同様に逆変換テーブルを作成する．実装は異なるが，仮想アドレスの列挙が必要であること変わらないため，`pas2vas` と同等の処理時間，メモリ使用量が要求されると考えられる．

メモリフォレンジックスツールである Volatility 3 [16] 向けにも `P2V` プラグインが公開されている [4]．当該プラグインは，Rekall の `ptov` に類似した実装となっているため，OS データ構造参照方式に分類できる．それ故に，当該プラグインにも `ptov` と同様の課題があると考えられる．

[*5] 一部のプロセスではユーザ空間が限定されており，`0x7ffe0000` が含まれていないものが存在した．

7. 議論

本章では，提案手法の制限と適用性について述べる．

7.1 制限

提案手法は，Rekall の ptov と異なり，未割り当ての領域に対して割り当て予定とされるページの情報を補完することはできない．提案手法では OS 独自のメモリ管理情報を解釈せず，ハードウェアが認識するページテーブルに基づいて逆変換を実現している．一方，Rekall の ptov は，OS のデータ構造に基づいて変換を行うため，割り当てが予定されているページの情報を補完できる．メモリ解析では，解析対象プログラムからアクセス可能であれば，実際にオンメモリであるかどうかによらず，仮想アドレスを得られる方が望ましい．そのため，現状の提案手法では ptov を完全に置き換えられないため，併用が必要となる．

7.2 他の OS やアーキテクチャへの適用性

5 章では，提案手法を Windows x64 向けの Rekall プラグインとして実装したものを用いて実験を行ったが，提案手法の適用範囲はこれに限定されない．前節で述べた通り提案手法は OS 独自のデータ構造を解釈しないが，それ故に他の OS 環境のメモリの解析にも適用することが可能である．また，x64 に限らず，同様のページングの仕組みがあるなら他のアーキテクチャに対しても使うことができる．ベースとするメモリ解析ツールも Rekall に限定されない．物理メモリへのアクセスや，仮想メモリ空間の再現，プロセスリストの取得などの基本的な解析機能を備えているものであれば利用することができる．

7.3 想定利用ケースへの適用性

提案手法は，2.2 節で述べた想定利用ケースへも適用可能である．すなわち，提案手法を用いた P2V 変換をベースとする共有メモリの検出や物理メモリスキャンの効率化が実現可能である．5 章で示した通り，提案手法は，高速さと正確性を両立している．全列挙方式のように日単位で結果を待つ必要もなく，逆変換で得られる仮想アドレスに OS データ構造参照方式のような制約もない．また，提案手法は OS 非依存でもあるため，OS 側に変更があっても安定して使用することができる．

8. おわりに

本論文では，高速かつ正確な P2V 変換を実現する新たな手法を提案した．提案手法では，ページテーブルを有向グラフに変換し，アドレスの変換を経路探索の問題に置き換えて解くことで P2V 変換を実現した．既存手法との比較評価を行い，それらと比べて高速かつ正確に変換可能であることを確認した．今後は，共有メモリを悪用するマルウェアの解析や物理メモリスキャンへの適用など，より実用的な状況を想定した実験を行い，性能を検証する．

参考文献

[1] Cohen, M.: Scanning memory with Yara, *Digital Investigation*, Vol. 20, pp. 34–43 (2017).

[2] Frisk, U.: GitHub - ufrisk/MemProcFS: MemProcFS, https://github.com/ufrisk/MemProcFS (accessed 2025-08-15).

[3] bata24: GitHub - bata24/gef: GEF - GDB Enhanced Features for exploit devs & reversers, https://github.com/bata24/gef (accessed 2025-08-15).

[4] Zohar, A.: GitHub - memoryforensics1/Vol3xp: Volatility Explorer Suit, https://github.com/memoryforensics1/Vol3xp (accessed 2025-08-14).

[5] Google Inc.: GitHub - google/rekall: Rekall Memory Forensic Framework, https://github.com/google/rekall (accessed 2025-08-14).

[6] Baranauskas, M.: NtCreateSection + NtMapViewOfSection Code Injection — Red Team Notes, https://www.ired.team/offensive-security/code-injection-process-injection/ntcreatesection-+-ntmapviewofsection-code-injection (accessed 2025-08-21).

[7] BreakingMalware.com: GitHub - BreakingMalware/PowerLoaderEx: PowerLoaderEx - Advanced Code Injection Technique for x32 / x64, https://github.com/BreakingMalware/PowerLoaderEx (accessed 2025-08-21).

[8] Ispoglou, K. K. and Payer, M.: malWASH: Washing Malware to Evade Dynamic Analysis, *WOOT 16* (2016).

[9] Narvaja, R.: Analysis of CVE-2022-21882: “Win32k Window Object Type Confusion Exploit” — Core Labs, https://www.coresecurity.com/core-labs/articles/analysis-cve-2022-21882-win32k-window-object-type-confusion-exploit (accessed 2025-08-21).

[10] Economou, N. and Nissim, E.: Getting Physical: Extreme Abuse of Intel Based Paging Systems, *CanSecWest 2016* (2016).

[11] Forshaw, J.: Project Zero: Raising the Dead, https://googleprojectzero.blogspot.com/2016/01/raising-dead.html (accessed 2025-08-21).

[12] Dolan-Gavitt, B.: The VAD Tree: A Process-eye View of Physical Memory, *Digital Investigation*, Vol. 4, pp. 62–64 (2007).

[13] Johnson, K.: KVA Shadow: Mitigating Meltdown on Windows — MSRC Blog — Microsoft Security Response Center, https://msrc.microsoft.com/blog/2018/03/kva-shadow-mitigating-meltdown-on-windows/ (accessed 2025-08-15).

[14] Mothe, R.: Randomizing the KUSER_SHARED_DATA Structure on Windows — MSRC Blog — Microsoft Security Response Center, https://msrc.microsoft.com/blog/2022/04/randomizing-the-kuser_shared_data-structure-on-windows/ (accessed 2025-08-21).

[15] Bellard, F.: QEMU, a Fast and Portable Dynamic Translator, *USENIX ATC 05* (2005).

[16] The Volatility Foundation: GitHub - volatilityfoundation/volatility3: Volatility 3.0 development, https://github.com/volatilityfoundation/volatility3 (accessed 2025-08-14).