コンパイラ出力コードモデルの尤度に基づくアンパッキング手法

rootkitのように，そしてゲノム解析のように

マルウェア対策人材育成ワークショップ 2008

岩村誠[*1,*2]，伊藤光恭[*1]，村岡洋一[*2]

[*1]NTT情報流通プラットフォーム研究所

[*2]早稲田大学

発表の流れ

- 研究の背景
- 従来のアンパッキング手法とその課題
- 提案手法
  - メモリアクセス監視手法
  - オリジナルコード特定方法
- 実験結果
  - CCC DATAset 2008のマルウェア検体
- まとめ

研究の背景

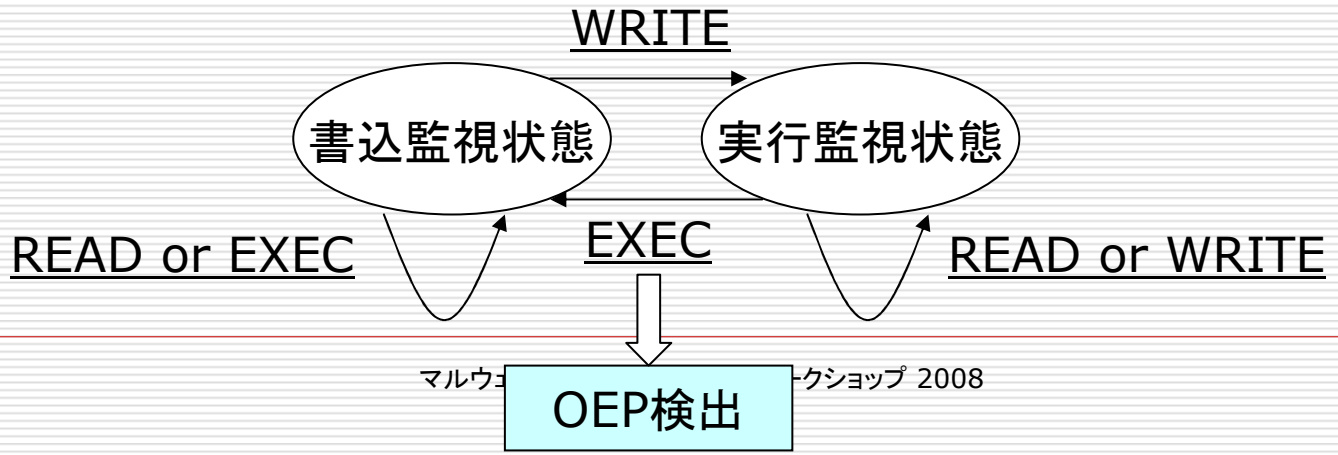
- マルウェアの種類数の急増
  - 最近では一日当たり数千検体の新種が収集される.
  - マルウェアのトレンドを掴むのも大変，対策の優先度決めも・・・
- 多数のマルウェアを効率的に管理・解析するには分類技術が非常に有用．マルウェアの分類技術は大きく2つに分けられる.
  - 挙動に基づくもの
    - アンパッキングや逆アセンブルを要さない.
    - ボットでは挙動を網羅的に把握することが困難.
  - プログラムコードに基づくもの
    - マルウェアの潜在的な機能を考慮した分類が可能.
    - ただ従来の分類技術では，
      アンパッキング済・逆アセンブル済であることが前提.

アンパッキングの自動化が重要

【従来研究】
・OllyBonE
・Renovo
・Saffron

Saffron（従来研究）

- 機能
  - 書込後に実行されたメモリ領域を検出することで，オリジナルエントリポイント（アンパッキング完了）を検出する．
- 実装
  - Windowsカーネルのページフォールトハンドラをフックする．
  - 全メモリページ(PTE)にSupervisor bitを設定する．
  - 全メモリページを書込監視状態（左）とし，発生したアクセス種別によって，状態を遷移させる．



Translation Lookaside Buffer (TLB)

- 特徴
  - PDE/PTE（物理アドレス変換用テーブル）専用のキャッシュ.
  - 命令フェッチ用とデータアクセス用，2種類のTLB（ITLB/DTLB）.
  - PTEを更新しただけでは，古いTLBが生きたままになる.
    通常はinvlpg命令等で明示的にTLBをフラッシュさせる.
- 応用
  - [Saffron]監視状態に対応するTLBのSupervisor bitを設定することで，監視対象のアクセスでPFを発生させる.
  - [Shadow Walker]同じリニアアドレスであっても，命令フェッチ時とデータアクセス時で別の物理アドレスを参照させることが可能になる.

余談

Saffronの場合（Supervisor bit）

	書込監視状態	実行監視状態
PTE	1	1
ITLB	0	1
DTLB	1	0

Shadow Walkerの場合

	Present bit	Phys. Page
PTE	0	-
ITLB	1	XXXX
DTLB	1	YYYY

Saffronの問題点

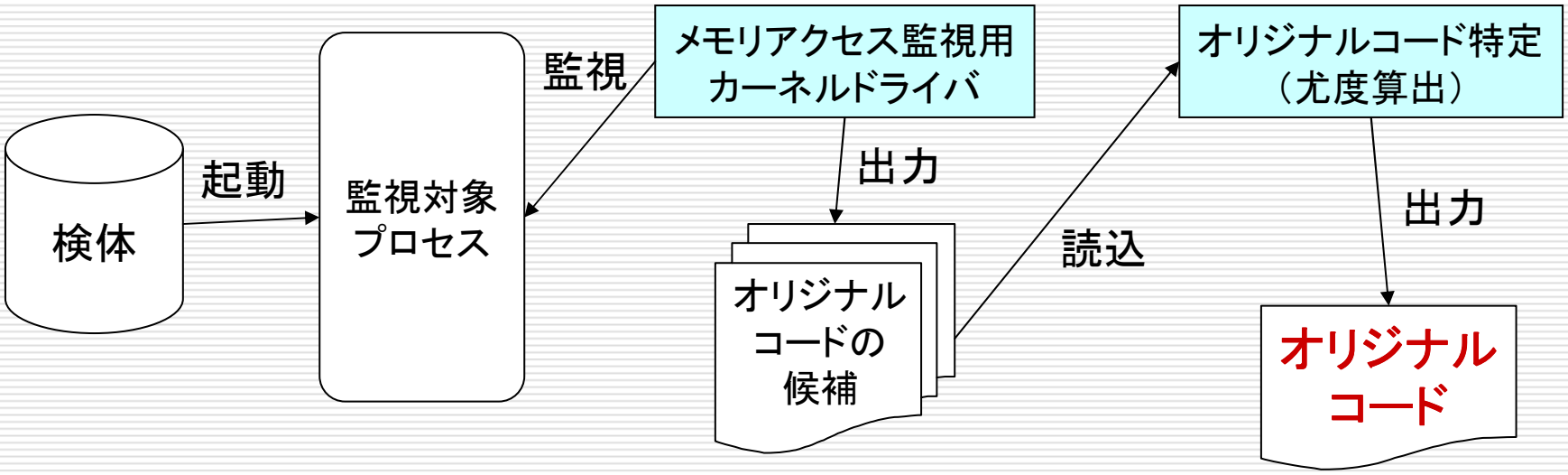
- 仮想マシン上で動作させる場合
  - ほとんどの仮想マシンには，**TLBが実装されていない**（もしくは，実機と実装が違う）．
    - VMwareの場合，特権モードをまたぐときに，ITLBがフラッシュされてしまう．
      →Supervisor bitをクリアした状態を作り出せない．
- 多重にパッキングされている場合
  - オリジナルコードとして，**アンパッキング途中のコードが検出**されてしまう．

提案手法

- TLBへの依存を無くし，**仮想マシン上で動作可能**にする
  Supervisor bitではなく，監視状態に応じたパーミッションの設定を行う
  - 書込監視時はread-onlyビットを設定
  - 実行監視時はXDbit (NXビット)を設定

【副産物】不要なPFが発生しなくなり，
解析速度もアップ↑

- 複数の候補から**オリジナルコードを特定**する
  - 詳細は後述.



オリジナルコードの特定

- 前提
  - オリジナルコードはコンパイラが出力したコード.
  - アンパッキングの処理ルーチンはコンパイラ出力コードっぽくない.
- 「コンパイラが出力したバイナリっぽさ」
  リバースエンジニアには分かる！？

```
  call  near ptr loc_2908+1
loc_2908:
  call  fword ptr [edi-7Fh]
  mov   dword ptr [ebp-1], 1F89FFFFh
  mov   [edi+4], edx
  pop   eax
  rdtsc
  push  ea
```

まだ
アンパッキング中
だな・・・

```
mov   edi, edi
push  esi
mov   esi, ecx
mov   eax, [esi+30h]
test  eax, eax
jz    short loc_100A41F
push  eax
push  40h
```

お？
それっぽい！

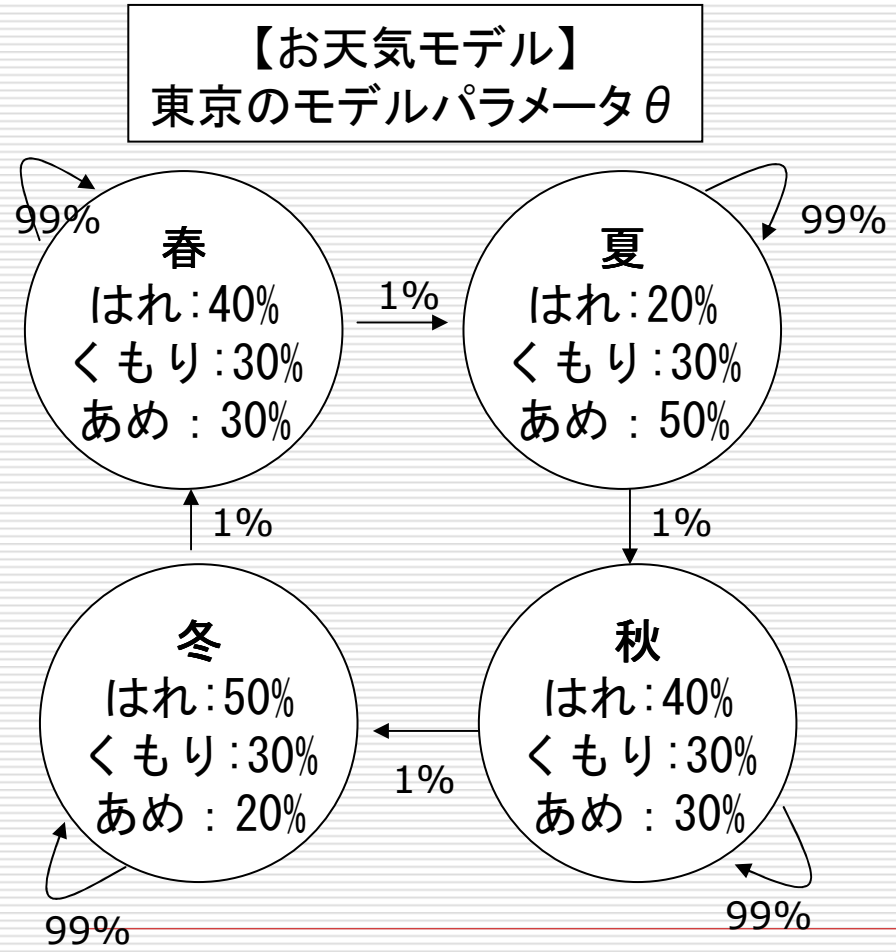
リバースエンジニアの脳みそ

- コンパイラ出力っぽい
  - よく使う命令（mov edi,edi）が出てくる
- コンパイラ出力っぽくない
  - あまり使わない命令（rdtsc）が出てくる
  - 命令の途中へ分岐（call loc_2908+1）してる

提案手法のアプローチ

命令の出現確率を隠れマルコフモデルでモデル化し，
「分岐命令の宛先は命令の先頭」という知見を盛り込みつつ，
コンパイラ出力コードモデルの尤度を算出する

隠れマルコフモデル



お天気系列
X={はれ，はれ，くもり，あめ，くもり}

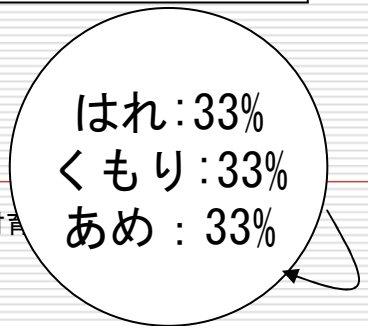
Forward algorithm

$$\sum_{all\ T} P(X,T \mid \theta) = P(X \mid \theta) = L(\theta \mid X)$$

尤度

東京のお天気パラメータが与えられたときの，
Xが起きる条件付確率．
Xが与えられたときのθの尤度，ともいう．

【nullモデル】

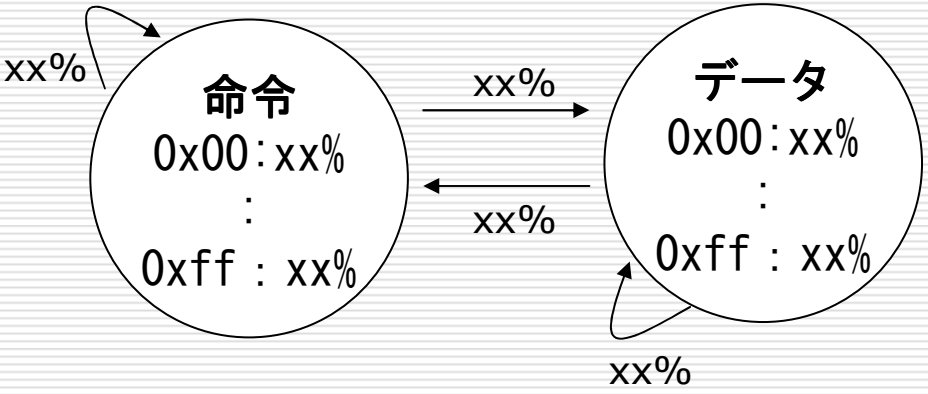


事前の知見がない
nullモデルと比較して，
L(θ|X)>P(X)なら
東京のお天気っぽい．

隠れマルコフモデル

【コンパイラ出力コードモデル】
モデルパラメータ θ は
既知のコンパイラ出力コードから学習

X={0xEB,0x01,0xFF,0xC3,…}



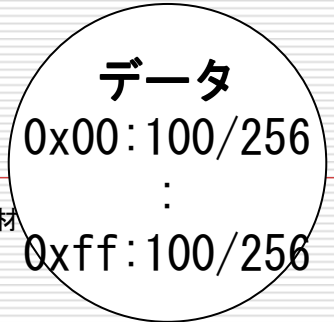
Forward algorithm

$$\sum_{all\ T} P(X,T\mid\theta) = P(X\mid\theta) = L(\theta\mid X)$$

尤度

コンパイラ出力コードモデルのパラメータが
与えられたときの，Xが起きる条件付確率．
Xが与えられたときの θ の尤度，ともいう．

【nullモデル】



事前の知見が
ないnullモデルと比較して，
L(θ |X)>P(X)なら
コンパイラ出力コードっぽい．

100%

「分岐命令の分岐先は命令」という制約

- たとえば以下のようなバイト列の場合
  - [1]0xEB[2]0x02[3]0x90[4]0x90[5]0xc3

通常のコンパイラが出力したコードであれば以下の条件を満たす.
・[1]が命令の先頭であれば[5]は命令の先頭
・[5]が命令の先頭でなければ [1]は命令の先頭でない.

分岐命令と解釈できる入力バイト値をx_i,その宛先をx_jとし，{i,j}の集合をB_xとしたとき
上記制約を満たしつつXが出力される確率は，次のように近似することでO(N)で計算可能

$$P(X, {}^{\forall}\{i,j\} \in B_X(t_i \neq \text{命令 } or\ t_j = \text{命令}) \mid \theta)$$

$$\approx P(X \mid \theta) \prod_{\{i,j\} \in Bx} (1 - P(t_i = \text{命令} \mid X, \theta) P(t_j \neq \text{命令} \mid X, \theta))$$

Nullモデル
と比較

実験概要

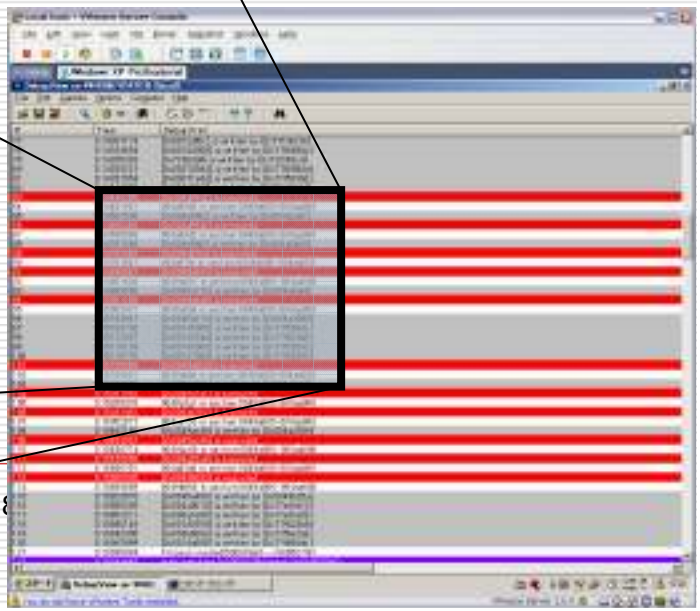
- 仮想OS上で研究用データセットCCC DATAset 2008のマルウェア検体のアンパッキングを試みた
- 仮想化ソフトウェア：VMware Server 1.0.4
  - ホストOS：Windows XP SP3
  - ゲストOS：Windows XP SP1
- ゲスト環境
  - CPU: Intel Xeon 2.66GHz x 1
  - メモリ：512MB
- モデルパラメータ θ
  - Firefox 3.0.1に含まれるxul.dllを利用
    MS C++により出力されたアセンブリを用いて学習
    データ状態はヌルモデル

実行画面

```
0.14444825      [0x004a9fc7] is written by [0x004a0d44]
0.14446305      [0x004a99d2] is executed
0.14451921      004a99d2 in section:0049a000-004aa000
0.14587665      [0x004a9f92] is written by [0x004a2a2f]
0.14590402      [0x004a99d2] is executed
0.14595598      004a99d2 in section:0049a000-004aa000
0.14761066      [0x004a9fd2] is written by [0x004a0b44]
0.14762463      [0x004a97fe] is executed
0.14767547      004a97fe in section:0049a000-004aa000
0.14848927      [0x0049a647] is executed
0.14851636      0049a647 in section:0049a000-004aa000
0.14999560      [0x0049a170] is written by [0x004a0af4]
0.15001208      [0x0049a6a8] is executed
0.15003471      0049a6a8 in section:0049a000-004aa000
0.15103847      [0x0049a214] is written by [0x004a3865]
0.15108736      [0x00145005] is written by [0x77f568dc]
0.15112367      [0x0014834d] is written by [0x77f523b6]
0.15116139      [0x00146000] is written by [0x77f52092]
0.15119770      [0x00147000] is written by [0x77f52092]
0.15155669      [0x0049a6bd] is executed
0.15159021      0049a6bd in section:0049a000-004aa000
0.15264902      [0x0049a22c] is written by [0x004a0af4]
```

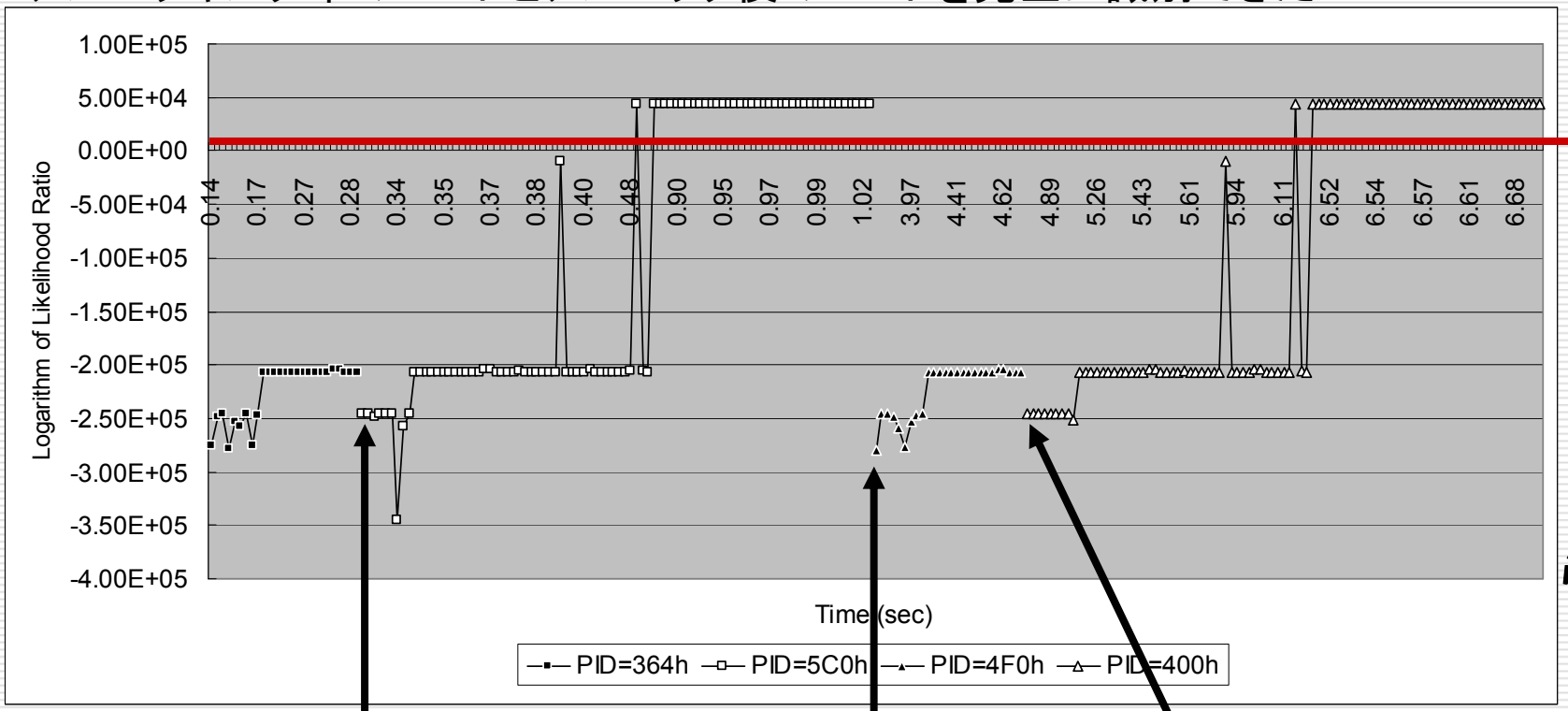
書込検出

実行検出



実験結果

約230のオリジナルコードの候補を抽出.
アンパッキング中のコードとアンパック後のコードを完全に識別できた.



尤もらしい

尤もらしくない

再起動1回目

再起動2回目

再起動3回目

最後に

- お話したこと
  - TLBの実装が不十分な仮想マシンにおいても，動作可能なアンパッキングのためのメモリアクセス監視機構の実現
  - 複数のオリジナルコードの候補から，オリジナルコードを特定する手法を提案・有効性を示した
- 今後の予定
  - さらに多くのマルウェアによる評価
  - プログラムコードに基づくマルウェア分類へ・・・

付録

従来のアンパッキング手法

- パッカー識別＆個別アンパッカー
  - PEiD等，シグネチャによるパッカーの識別
  - パッカーに応じた個別アンパッカーの利用
    e.g., upx –d
  - 以下のようなケースに対応できない．
    - パッカー識別を騙すマルウェアの存在
    - 新たなパッカーの出現

- 汎用的なアンパッカー
  - メモリ監視により，「書込み後，実行される」領域をとらえることで，マルウェアのオリジナルコード（**動的に生成されたコード**）を検出する．使われているパッカーに依存しない．
    OllyBonE, Renovo, Saffron, …

Intel x86におけるページング機構
リニアアドレスから物理アドレスへの変換