静的解析と動的解析を組み合わせたバイナリーコードの制御フロー解析

IIJ-II 技術研究所
泉田大宗

背景

- 現在もなお多数の正体不明なバイナリプログラムがネット上を飛び交っている
- 正体不明
  - ソースコードがない
  - 誰がどのように作ったかもわからない(使用したコンパイラなど)
  - 暗号化、難読化など
- 対策としては
  - アンチマルウェアソフト(シグネチャマッチ)
  - セキュリティ教育
- 振舞を調べるには
  - サンドボックス環境で実行してトレースを解析
  - すべての実行パスを調べられるわけではない

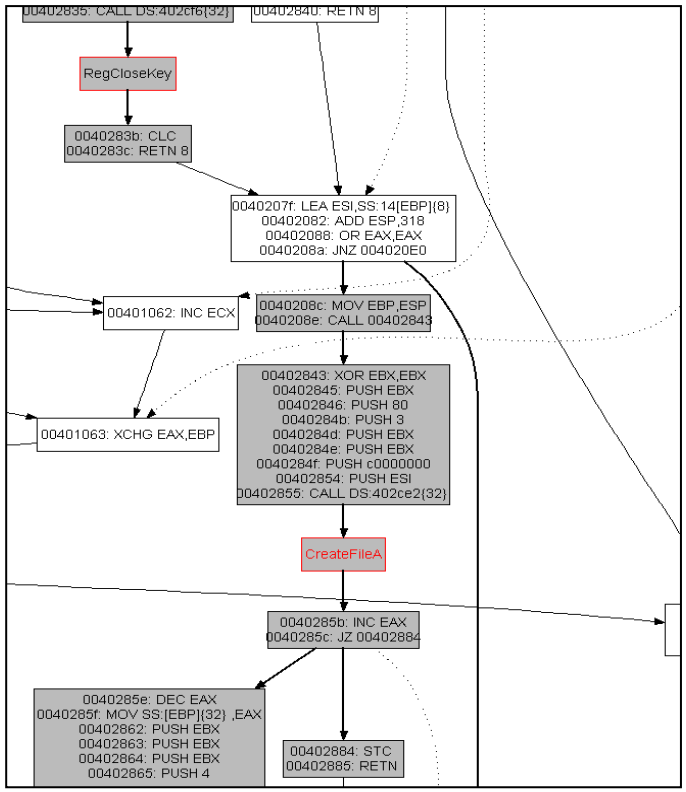
制御フローの再構築

- バイナリプログラムから、最低限の知識だけを用いて制御フローグラフを再構築する

バイナリプログラム

```
fcbe8610400056b8002040005097b280bb701040
00a4ffd373fb33c9ffd3731433c0ffd3731d41b010ff
d312c073fa7535aaebe2e84300000049e210e839
000000eb21acd1e8743d13c9eb159148c1e008ac
e82300000080fc0573068 3f87f77024141958bc55
68bf72bf0f3a45eeba602d275058a164612d2c333
c941ffd313c9ffd372f8c3e87002c002ebfe33d2643
9ff328f8922be272c4003bfba180be809334b4539
524e8e4c333201ff1534300093e8536f082ae0c0a
c91e305005603f1ebe983ec5473817a18cf288b26
e82a6100536f6674776172650e5c4d6963df1b73
1dd0574142a10434800a61622046696c7565794e
756de06ac4c27807008d751481c4a0410bc0f454c
c8bb02972794b004e64e34603766051f2c61c807
e01007512568d7d0c00576a485966adaae200fb5
85e83c62055e866076d5d14022459e2d8ff758ea1
6c040608a1a7be55bb38068303c45464678f0619
56586a25c2133a000269776f726d2e61786c387a
65f73d62797ca1b330c46e21fe6b1c78326f59314
30dc47768a4ab1d747970cd6e670e9b1a730afb7
8ffafdf28ddaa38961431676f086164c7df3
```

制御フローグラフ（CFG）



CFGがあると?

- プログラムの振舞に対する網羅的解析が可能
  - 既存の静的解析の技術を適用: 抽象実行、モデル検査など
- グラフから特徴ベクトルを抽出して自動分類
  - 機械学習など

しかし、

CFGを作ること自体が難しい!

難しさ その1

- 間接ジャンプ命令

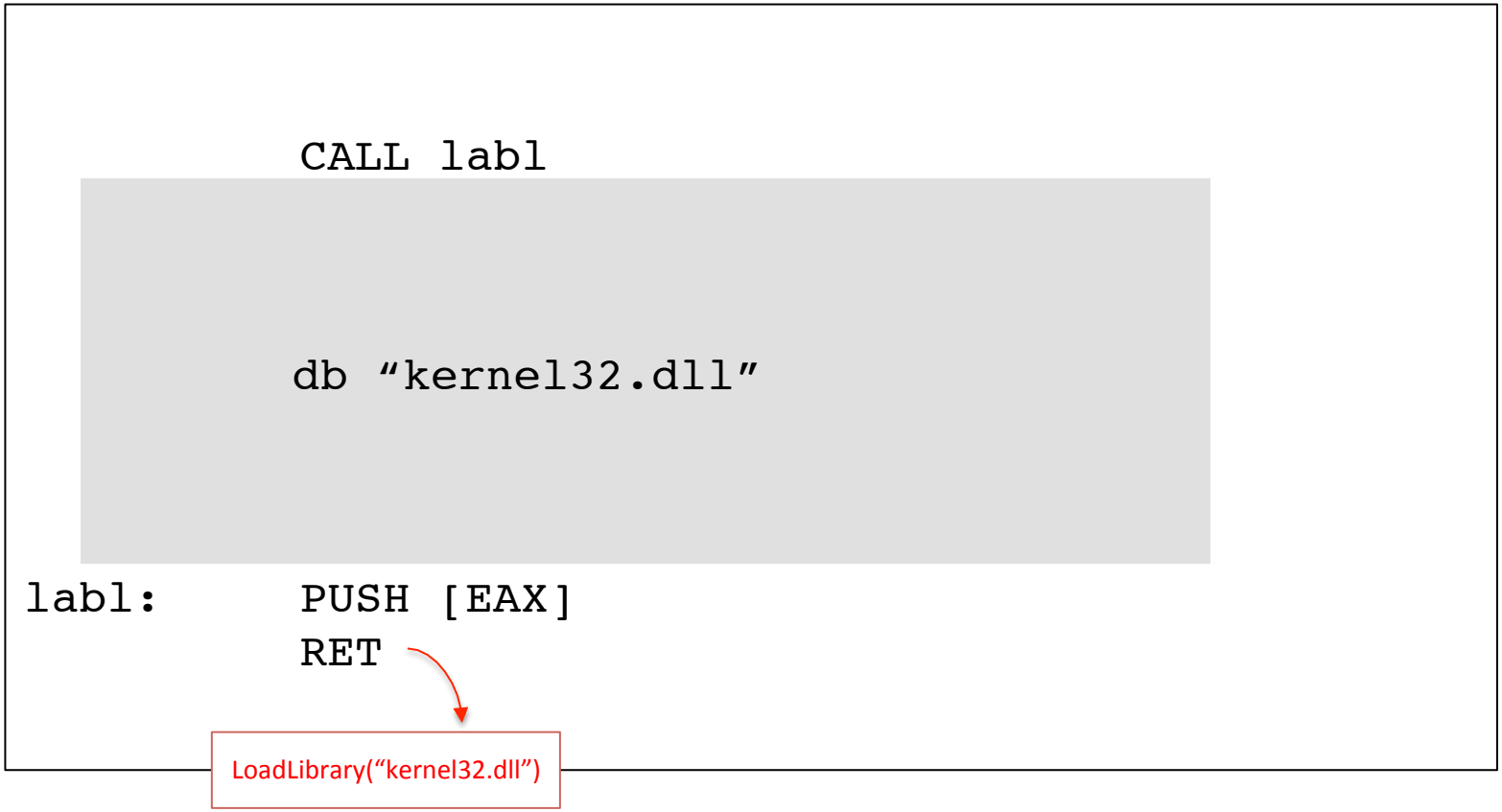
```
JMP　[EAX]
RET
```

- レジスタやメモリ状態を決定するためには値解析が必要
  - でも、値解析にはCFGが必要
- 一般に静的には決定不能

難しさ その2

- CALL/RET命令が信用出来ない!

CALL/RET

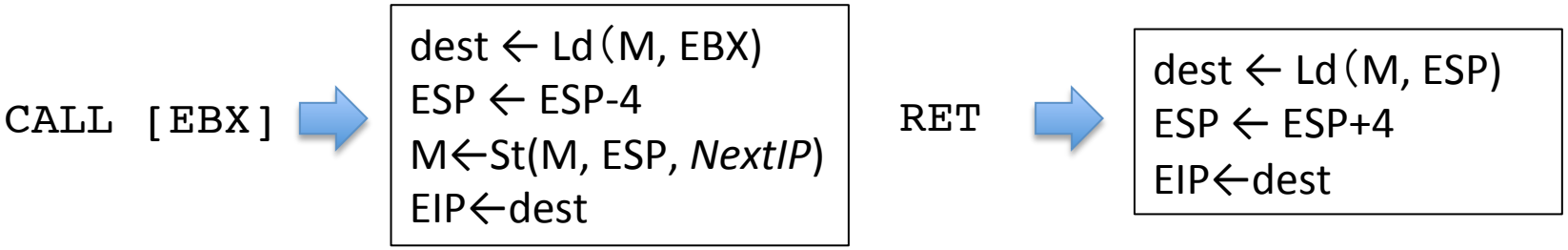

CALL labl
db “kernel32.dll”
labl: PUSH [EAX]
RET
LoadLibrary(“kernel32.dll”)

難しさ その2

- CALL/RET命令が信用出来ない!
  - 標準的なコンパイラを使用しているとは限らない
  - CALL/RETが関数呼び出し/復帰となっているとは限らない
  - 従って、事前にプログラムから関数を切り分けることはできない
- 他にも暗号化、難読化、自己書き換えなど

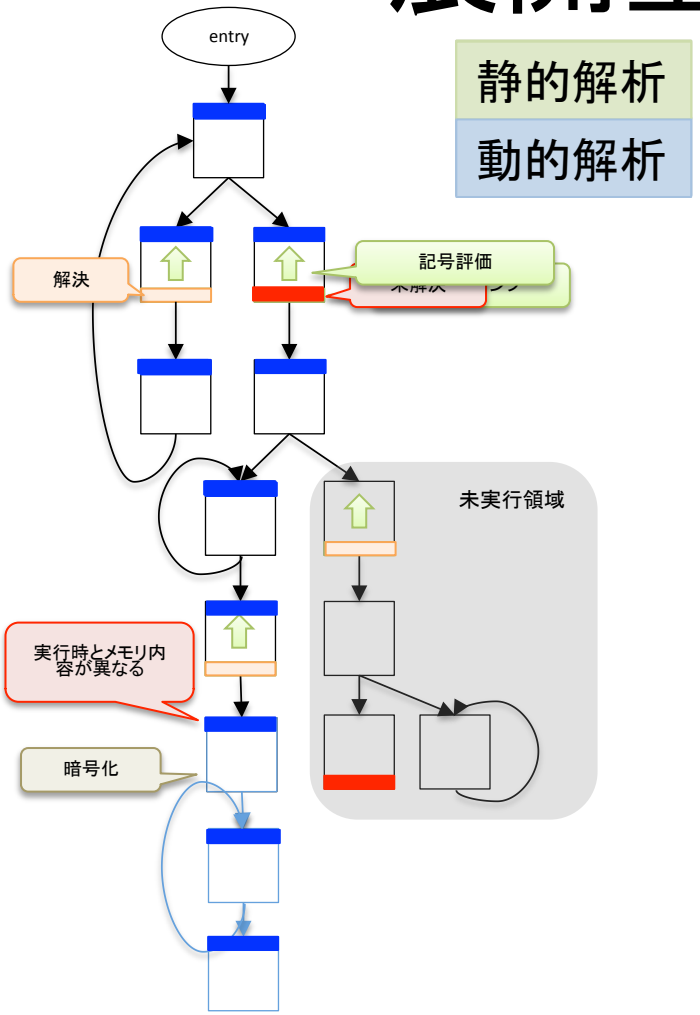
対策

- 各機械命令を低レベルな記号式で解釈して記号評価
- スタックを抽象化せず、ランダムアクセスメモリとして解釈



- プログラム全体解析(部分構造に分解しない)
- 展開型制御フロー解析

展開型制御フロー解析



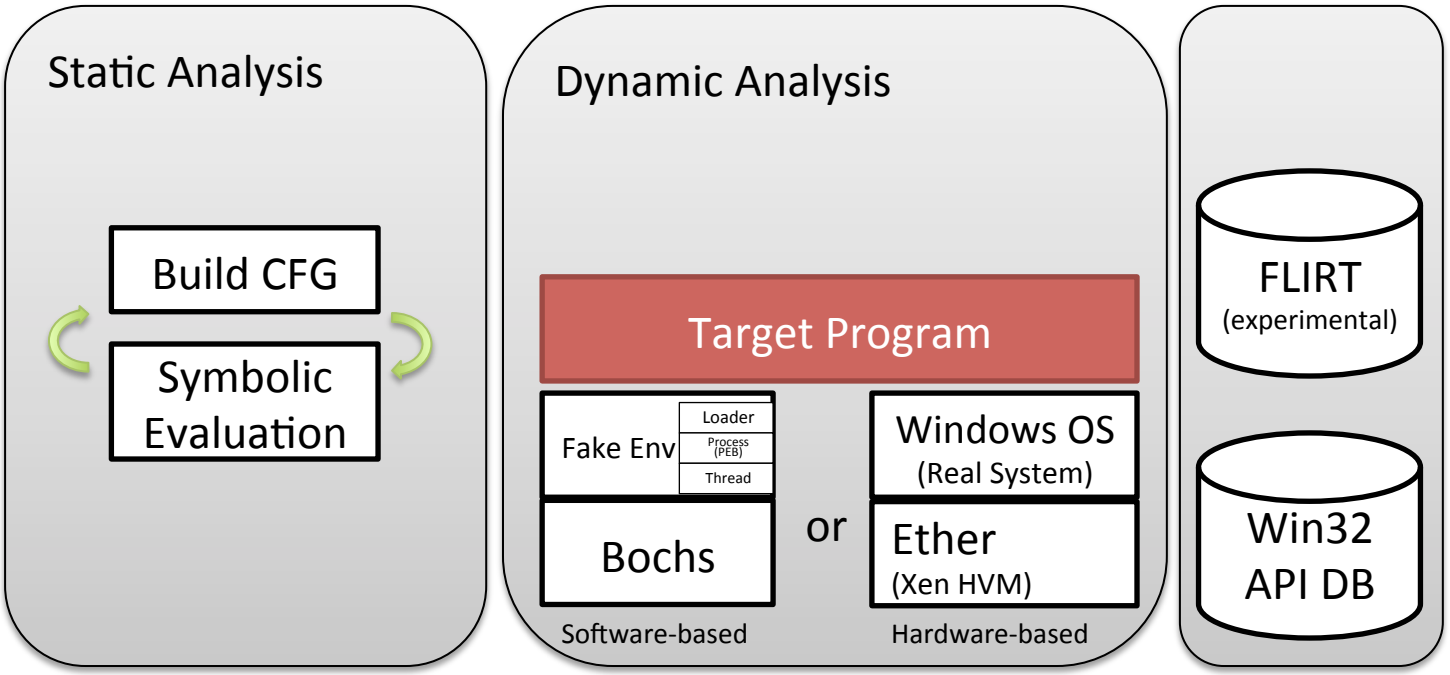
1. 静的解析を開始(CFG作成)
   a. 直接ジャンプを辿ってできるだけ展開
   b. 間接ジャンプの行き先を記号評価を用いて解決
   c. もし定数アドレス値に定まったらその先をさらに展開
   d. 解決可能な間接ジャンプがなくなるまで繰り返し
2. 動的解析を開始
   a. 作成されたCFGを参照しながら動的実行
   b. もしも、
      i. 実行がCFGをはみ出した
      ii. 実行中にコードが書き換わった

      場合はそこから静的解析を行う
3. 実行が終了するまで繰り返す

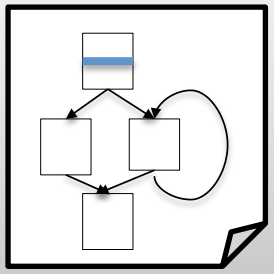
ここまでのまとめ

- 静的解析でできるだけ頑張ってCFGを作成する
- 静的には解決できない間接ジャンプのために動的解析を利用する
- 一回の解析プロセスで静的/動的を切り替えるので、
  - 静的解析は動的な情報を参照可能
    - 変更があったメモリ、途中で読み込まれたDLLなど
  - 動的解析は(その時点での)CFGを参照可能

システム概観



Controller



- Single Step Execution
- Memory Modification Check
- Build New CFG if any discrepancy found

出力例 (Win32/Rustok.B)

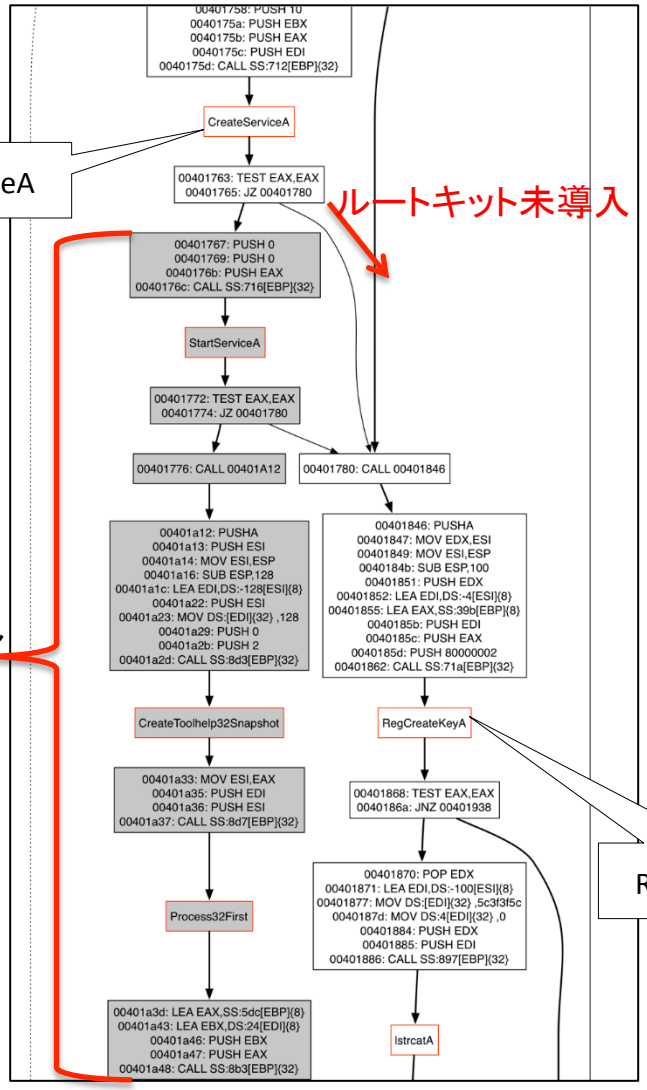
復号化

APIテーブル作成

実行されなかったブロック

ルートキット導入

CreateSerivceA



実行トレース

```
OpenSCManagerA('', '', 983103)
CreateServiceA(0, 'pe386', 'Win23 lzx
files loader', 16, 1, 1, 0, '','Base',
0, '', '', '')
RegCreateKeyA(2147483650L, 'system\
\CurrentControlSet\\Services\\lzx32',
134580)
lstrcatA('\\??\\', '')
RegSetValueExA(16, 'ImagePath', 0, 1,
134328, 4)
RegSetValueExA(16, 'DisplayName', 0, 1,
4201371, 21)
RegSetValueExA(16, 'ErrorControl', 0, 4,
4200764, 4)
RegSetValueExA(16, 'Start', 0, 4,
4200768, 4)
RegSetValueExA(16, 'Type', 0, 4,
4200772, 4)
RtlInitUnicodeString(134328,u'\\registry
\\machine\\system\\CurrentControlSet\
\Services\\lzx32')
ZwLoadDriver(134328,)
ExitThread(0,)
```

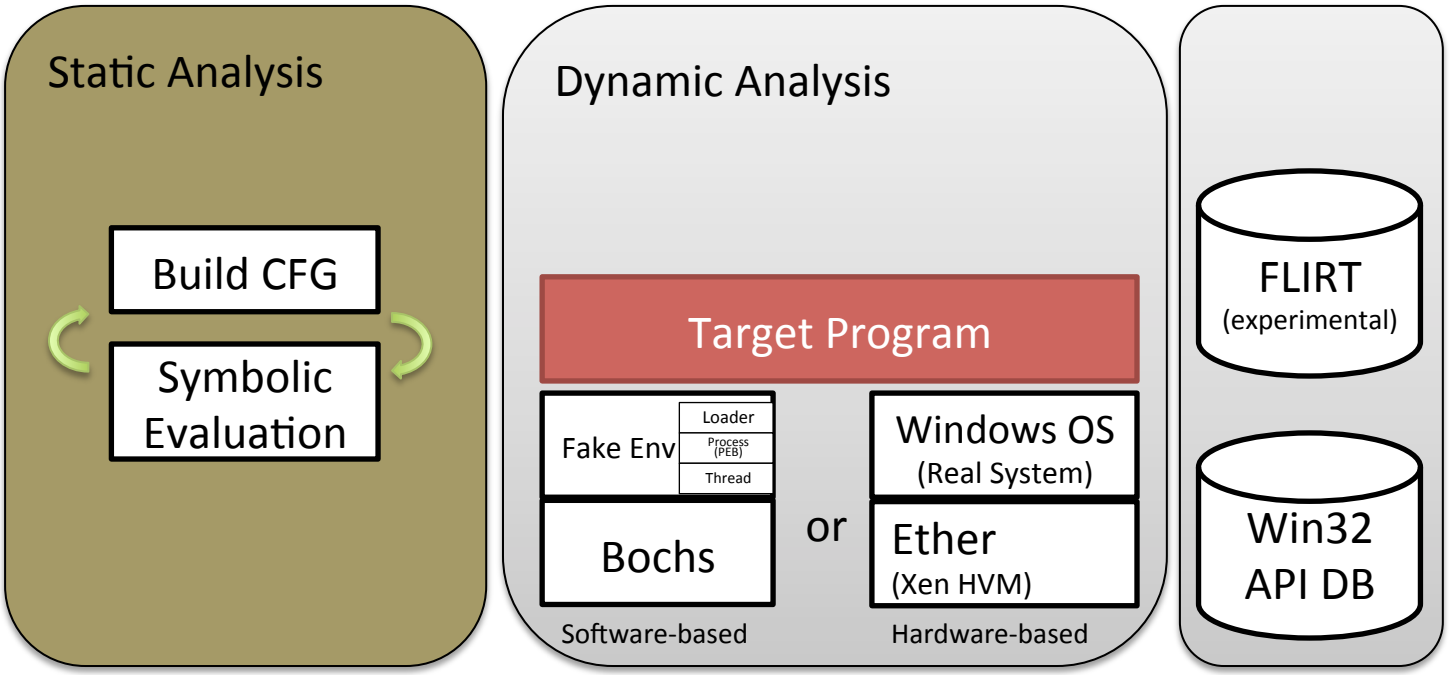
RegCreateKeyA

Node: 253 Edge: 284

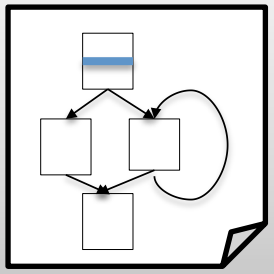
Time: 722s(Static) 392s(Dynamic)

Indirect Jump: 114(3 unresolved)

静的解析について

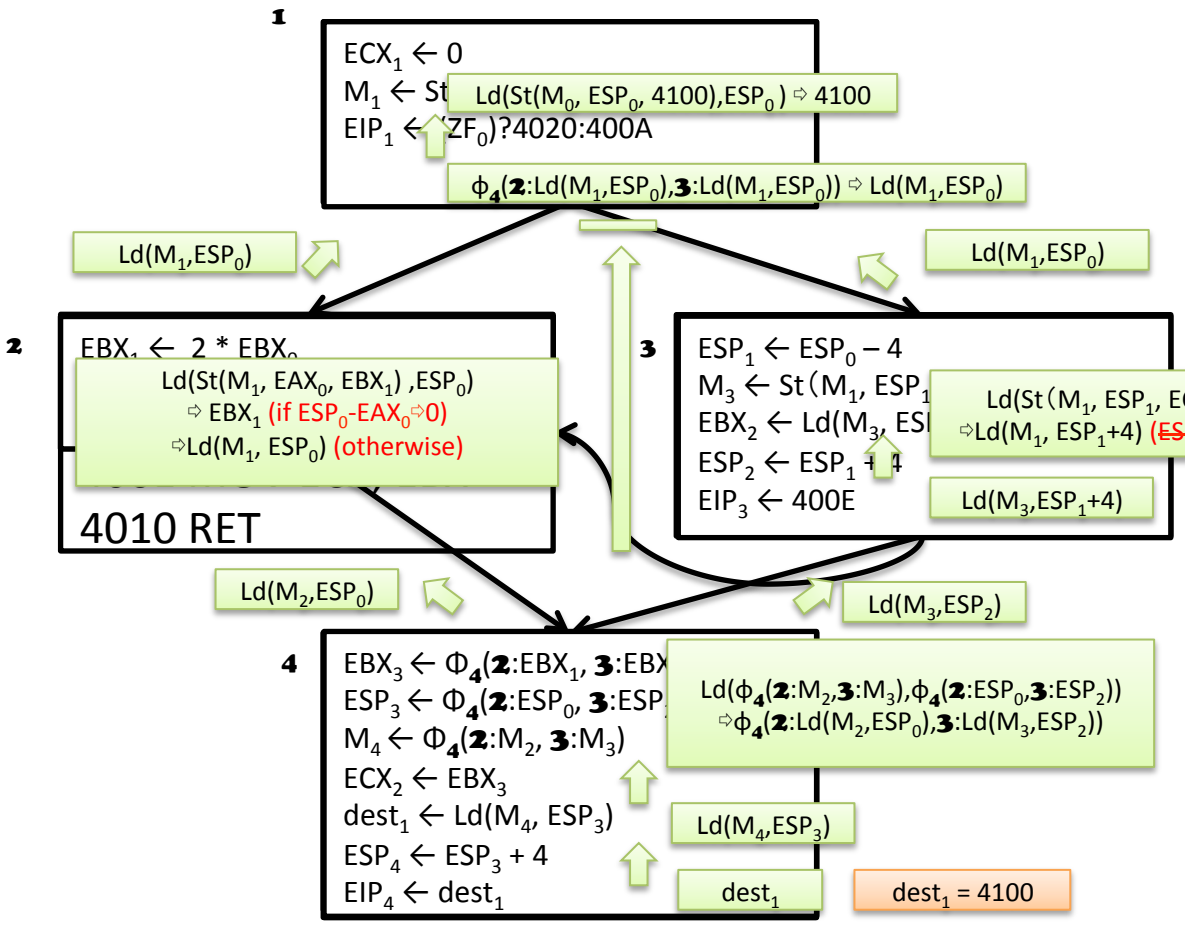


Controller



- Single Step Execution
- Memory Modification Check
- Build New CFG if any discrepancy found

記号評価



1. 直接ジャンプを辿ってCFGを展開
2. 各命令を単純な代入文列に変換
   - メモリ状態変数: M
   - 読出: Var←Ld(M,Addr)
   - 書込: M←St(M,Addr,Val)
3. CFGを静的単一代入(SSA)形式
   - 定義式が一意となるように変数名を変更
   - Φ関数
4. 要求駆動型定数伝播
   - 各変数をその定義式で置換
   - Use-def鎖をたどる
   - 記号代数演算

Φ(A,A) ⇨ A
f(ϕ(A,B), ϕ(X,Y)) ⇨ ϕ(f(A,X),f(B,Y))
Ld(St(M,A,V), A')
⇨ V (if A = A')
Ld(M,A') (otherwise)

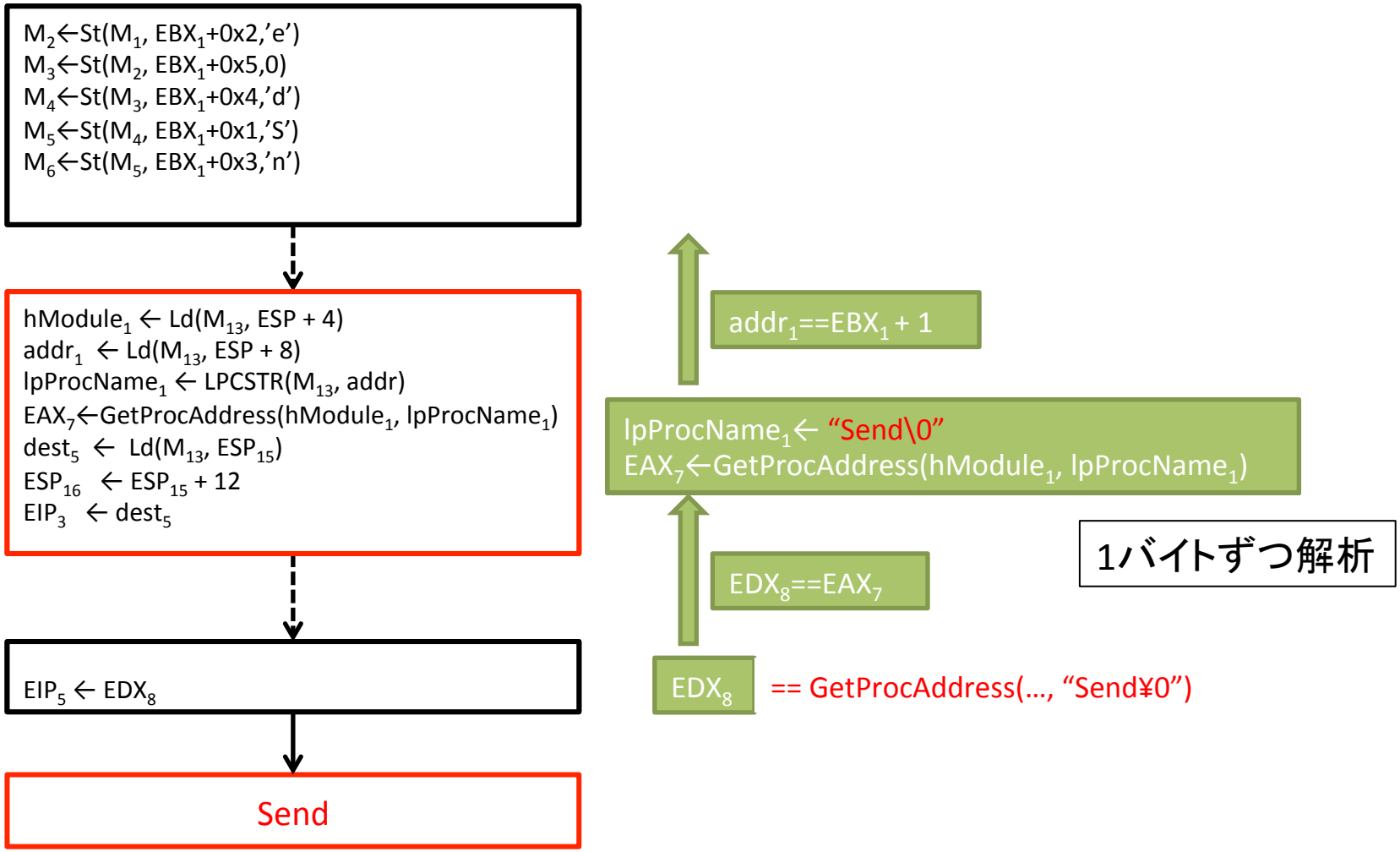
API ブロック

- 行き先アドレスがWin32 API関数のエントリアドレスだった場合、APIブロックを作成
  - スタックの巻き戻し
  - 返り値を計算する
  - 必要あれば文字列の評価も

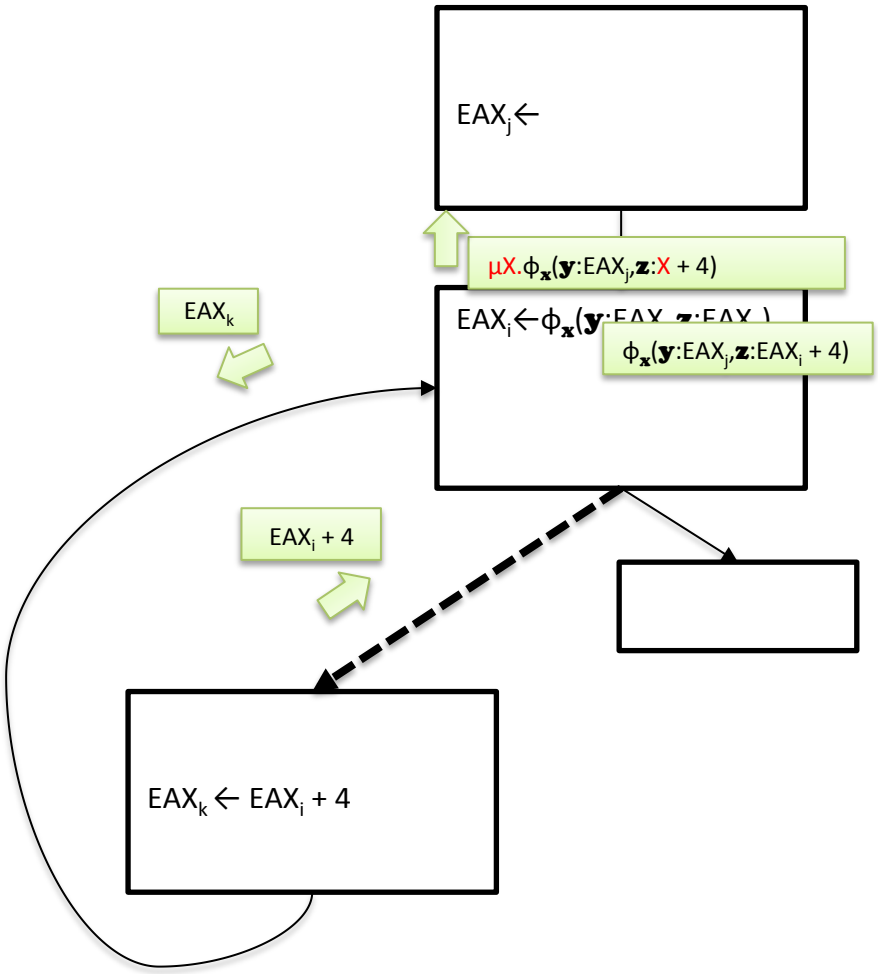
GetProcAddress

```
hModule ← Ld(M, ESP + 4)
addr  ← Ld(M, ESP + 8)
lpProcName ← LPCSTR(M, addr)
EAX ← GetProcAddress(hModule, lpProcName)
dest  ←  Ld(M, ESP)
ESP   ← ESP + 12
EIP   ← dest
```

文字列解析

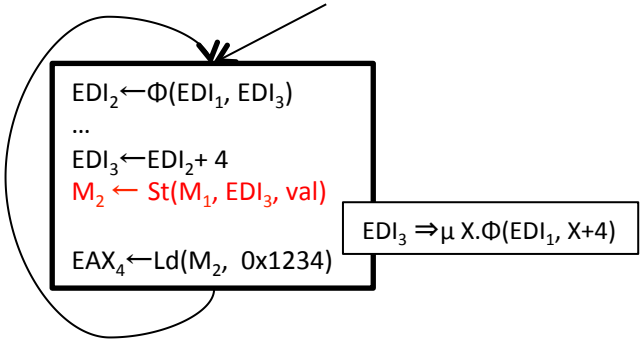

M_2←St(M_1, EBX_1+0x2,'e')
M_3←St(M_2, EBX_1+0x5,0)
M_4←St(M_3, EBX_1+0x4,'d')
M_5←St(M_4, EBX_1+0x1,'S')
M_6←St(M_5, EBX_1+0x3,'n')
$hModule_1$ ← Ld(M_{13}, ESP + 4)
$addr_1$ ← Ld(M_{13}, ESP + 8)
$lpProcName_1$ ← LPCSTR(M_{13}, addr)
EAX_7←GetProcAddress($hModule_1$, $lpProcName_1$)
$dest_5$ ← Ld(M_{13}, ESP_{15})
ESP_{16} ← ESP_{15} + 12
EIP_3 ← $dest_5$
EIP_5 ← EDX_8
Send
$addr_1$==EBX_1 + 1
$lpProcName_1$← "Send\0"
EAX_7←GetProcAddress($hModule_1$, $lpProcName_1$)
EDX_8==EAX_7
EDX_8
== GetProcAddress(…, "Send¥0")
1バイトずつ解析

ループ

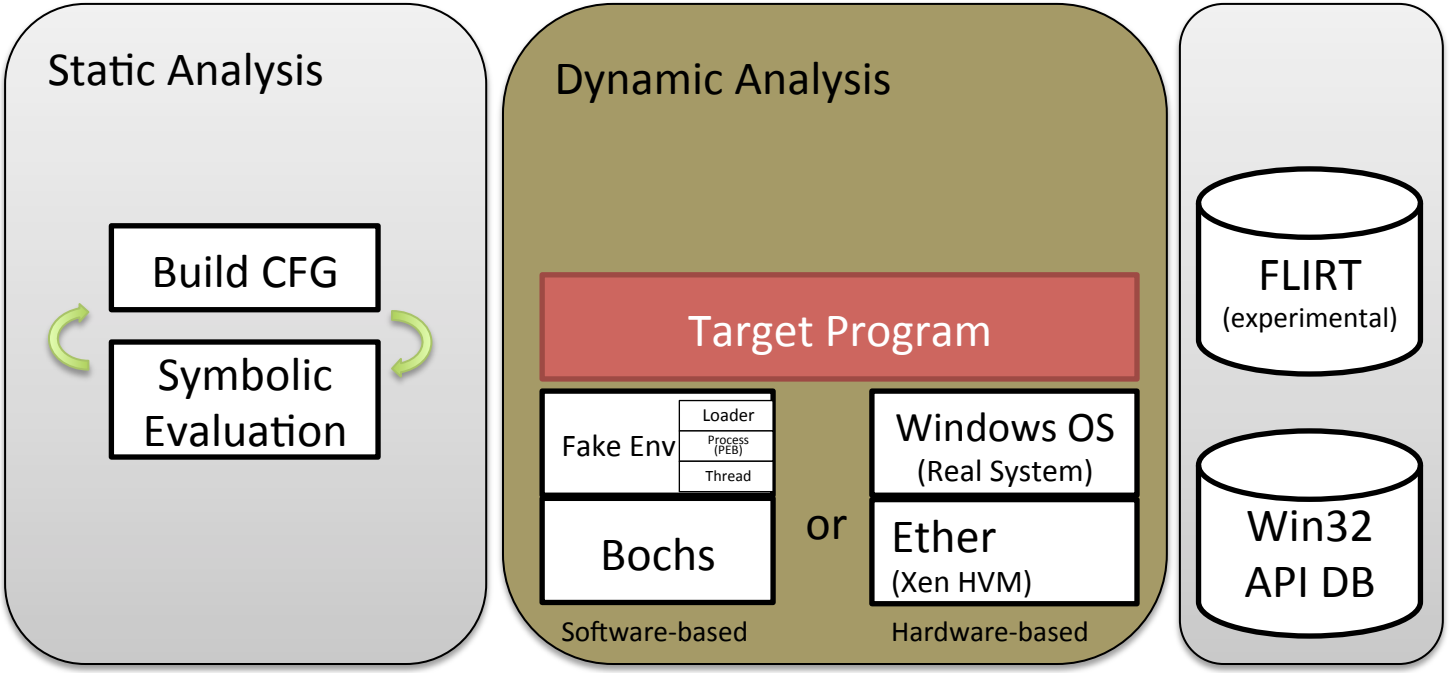


各定義式 $V_i \leftarrow \phi_{\mathbf{x}}(\mathbf{y}:V_j, \ldots)$について

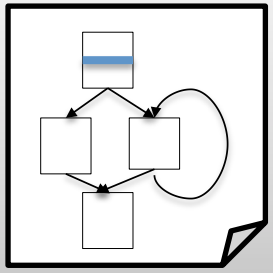
- ループ辺に沿った評価を行う
- もし、V_iが結果に再び現れる場合は、新しい変数Xに置き換えて不動点演算子μXを導入
- ループ辺以外の評価を行う.
- ループ不変が自明のときだけ簡約
  μX. Φ(X, A) ⇨ A (loop invariant)
- アドレス比較において不動点演算子が現れた場合は、その時点で比較失敗とする



動的解析について



Controller



- Single Step Execution
- Memory Modification Check
- Build New CFG if any discrepancy found

動的解析に求められるもの

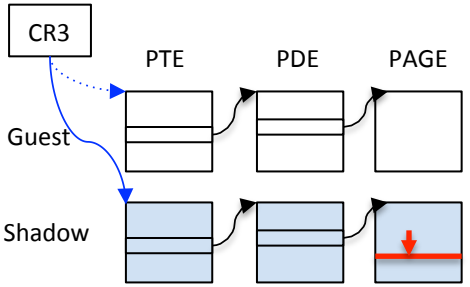
シングルステップ実行	• 1命令ごとに実行 • CFGとの違いをチェック
ブレークポイント実行	• 不必要なところはスキップ • ライブラリ関数内など
静的解析への 実行時情報の提供	• メモリ上にロードされたコード • DLLのインポートテーブルなど
書き込み検知	• 復号化の検知 • 自己書き換えコードの検知
ステルス性	• ターゲットプログラムから実行環境を検知されない

ソフトウェアベース動的解析

- Bochs-python (E.Carrera)がベース
  - Bochsのデバッガインターフェースをフックして Pythonインタープリタを起動
  - Bochsの内部情報にアクセスするための Pythonインターフェース
  - CPUとメモリ部分だけ使用
- 軽量な擬似実装環境
  - PEファイルローダ
  - 実行時情報(TEB, PEBなど)
  - Win 32 API のスタブ実行
    - ヒープ管理、ファイルシステムのエミュレート
  - ターゲットにWindows上で実行されていると勘違いさせるための最低限度の実装

ハードウェアベース動的解析

- Ether(ジョージア工科大)がベース
  - Xen 3.1を改変したイントロスペクションツール
  - SYSENTER命令を監視してシステムコールのトレース(システムコール名、引数、返り値)を出力
  - システムコールの終了を検知するためにページフォルト処理を用いたブレークポイントを利用
    - BPの置かれたシャドウページをはずす
    - PFが起きた場合
      - RWなら、一旦戻して、すぐはずす
      - Xで、BPの近くならそのままステップ実行



比較

	ソフトウェアベース	ハードウェアベース
シングルステップ	CPU LOOPを適時回数実行	TFを使用
ブレークポイント		シャドウページ
実行時情報	Python APIを通して読み出し	Guestページを読み出し
書き込み検知	Dirty Bitを使用	xc_shadow_control
ステルス性	• APIスタブを用意すれば環境はばれにくい • CPU シミュ自体は検知可能	• Guest側から検知することは困難
特徴	+ 軽い(並列化可能) - 環境のエミュレートが完全ではない	+ 実環境に近い実行 - 重い
問題点	• スタブを用意するのが大変	• 不安定!(特にブレークポイント)
	コードベースが古い!	

今後の課題

- 現在はProof Of Conceptの段階
- 様々なアーキテクチャや実行モデルへの対応
  - Linux
  - シェルコード、デバイスドライバなど
- ソフトウェアベース動的解析
  - 静的解析の結果を利用して振舞を変更?
- ハードウェアベース動的解析
  - Etherベース → Drakvuf/Libvmiベース

Drakvuf(Tamas K. Lengyel)

- エージェント無しでマルウェア実行
  - Sandbox側に特別なアプリをインストールしなくていい
- エージェント無しでカーネルの関数呼び出しをモニター
  - モニタ用DLLをインジェクトする必要無し
- ファイルやヒープの生成をトレース
  - ファイルが削除される前に内容をコピー
- VMをCopy-on-writeなクローンを作る
  - スケーラビリティ向上

Drakvufの構成

- Libvmi (Xen backend)
  - XenAccessの後継
  - XC(Xen Control)ライブラリに依存
  - イベントマネージャ(Register, Memory, Interrupt, SingleStep)
  - Python用API: pyvmi
- Volatility
  - Memory Forensicツール(Windows/Linux/Mac OSに対応)
  - Pythonで書かれている
  - メモリアクセス用のプラグインとしてpyvmiを用いたものがある(Libvmiが配布)
  - Drakvufではファイルの書き出しに使用
- Rekall
  - Volatilityからのブランチプロジェクト(by Google)
  - DrakvufおよびLibvmiではPDB情報からNTカーネルのシンボルテーブルを作成するのに使用

プロセスインジェクション

- 親プロセスのPIDを指定
  - vmi_pid_to_dtb: PID→DTB(CR3)
- CR3の変更を監視(Register Event)
  - 親プロセスのCR3を見つける
    - KPCR→PCRB→CurrentThread
    - ユーザランドの復帰ポイントにブレークポイント
  - ユーザランドに戻ったら、スタックとレジスタを保存して、強引にCreateProcessAを呼び出す
  - プロセスの作成が終わったらスタック、レジスタを戻す

ブレークポイント

- ブレークポイントを置きたい物理アドレスに0xCC(INT3)を書き込む(バックアップを保存)
- メモリアクセスを監視(Memory Event)
  - ブレークポイントを置いたアドレスの読み出し/書き込みがあれば、一旦元に戻し、1ステップ実行後にまたINT3を書き込む
- 割り込みを監視(Interrupt Event)
  - 現時点で監視可能な割り込みはINT3のみ
  - ブレークポイントを置いた場所で、なおかつ監視対象のプロセスであれば、ブレークポイントを除いて適時処理
  - 監視対象でないプロセスの場合は、一旦元に戻してから、1ステップ後にまたINT3を書く

静的解析との連携

- Drakvufをそのまま使うのは難しい
  - 現在はNTカーネルの特定のAPI呼び出しを監視するようにハードコードされている
- 使えそうなのは
  - ブレークポイント機構
  - プロセスインジェクション
  - くらい？
- 実装上の問題
  - PyvmiはイベントAPIを実装していない
    - なので、追加してみた(実験中)
  - Pythonで同等の機能を書くことはできそう
  - シングルステップまわりがちょっと大変?
    - プロセスの切り替えを監視して、対象以外はシングルステップをOff
    - 対象プロセスでも、カーネルランドなどはOff
  - Pythonで書ければ、VolatilityやRekallなども使える?

おわり

何か良いアイデアやシステムがありましたら、
いつでも教えてください。

関連研究

- 静的バイナリーコード解析
  - CodeSurfer/x86 [Balakrishnan, Reps]
  - BAP (formerly known as Vine) [Brumley et al.]
  - Jackstab [Kinder]
  - BE-PUM [Nguen, Quan, Ogawa]