

# Dealing with Virtualization packer

Boris Lau, SophosLabs

CARO 2008, Amsterdam



## Define:obfuscator

“A program transformed into an equivalent but visibly different run-time package, which when used operates as if in its original form”

Paul Ducklin

# Dealing with Virtualization obfuscators

Boris Lau, SophosLabs

CARO 2008, Amsterdam



## Aim

- Theoretical research into ways of scanning through Virtualization obfuscators
- Synopsis
  - Introduction
  - Analysis case studies
  - Designing Detection
  - Technicality with detection

# Introduction

What is a virtualization obfuscator?

Introduction

Analysis case studies

Designing Detection

Technicality with detection

# What is Virtualization?

- Commonly used in many fields
  - Resource virtualization – e.g. Virtual Memory
  - CPU virtualization – e.g. VMware, VirtualPC
  - Application virtualization – e.g. Java Bytecode, .NET CIL
- Virtualization is an abstraction of an existing interface <sup>(1)</sup>
  - Normally to provide some extra features
- Virtualization can simplify or complicate analysis

(1) Eric Traut, Distinguished Engineer and Director of Kernel & VM Development, Machine Virtualization, University of Illinois ACM computing conference, 2007

## Virtualization used as an obfuscation technique

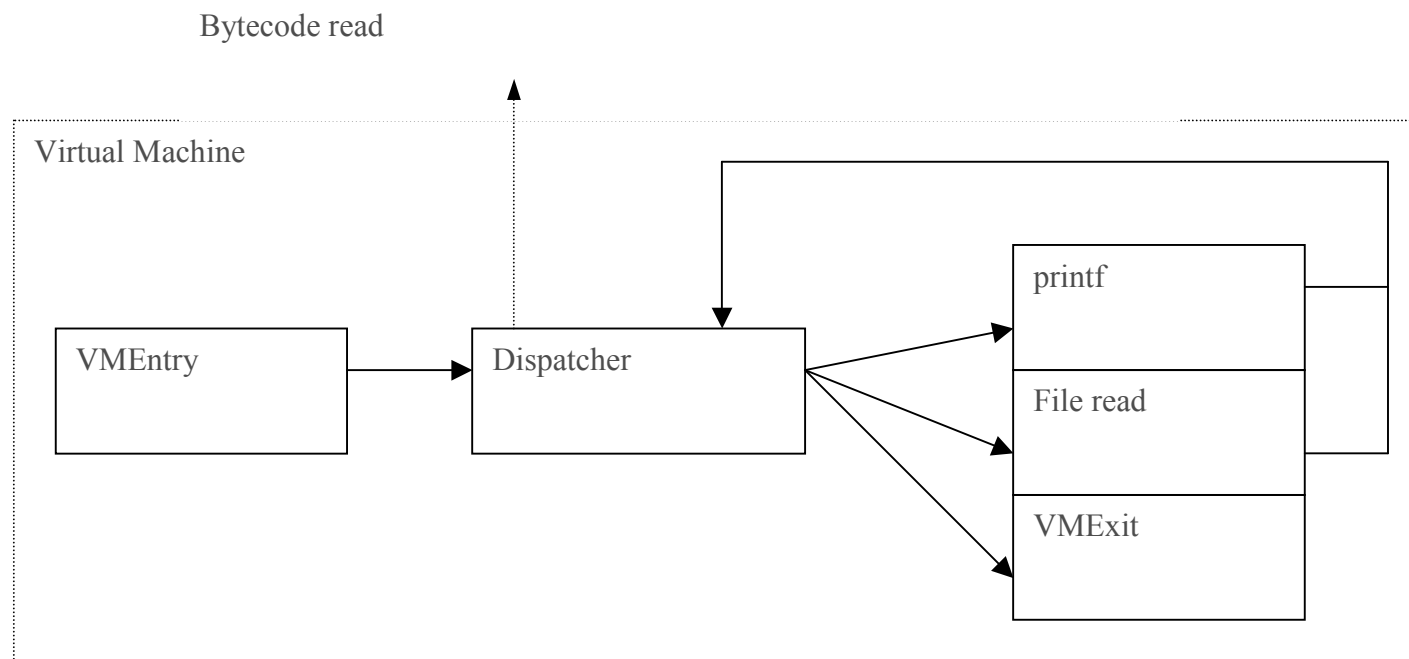
- Making it more difficult to understand
  - Opportunity to introduce extra complexity
  - Render reverse engineer's native knowledge useless
- Original code never reappears at execution time
- Used by commercial obfuscators
  - For malware as well as legitimate applications

## Definitions

- Virtual machine (VM)
  - a software implementation of a machine (computer) that executes programs like a real machine<sup>(1)</sup>
  - **Dispatcher** – part of the VM which read the bytecode and handles control flow logic
  - **Handler** – part of VM which carry out the execution of bytecode
- Bytecode
  - Binary interprets by the VM which determines the behaviour of the sample
- VM context
  - data modified during execution of virtual machine

(1) [http://en.wikipedia.org/wiki/Virtual\\_machine](http://en.wikipedia.org/wiki/Virtual_machine)

## Typical architecture of a VM



# Decoding a handler

## •What does this do?

### •Case 1

```
Stack esp+0 : 0012ff94 => 00000001
eax: 0000c541 => 00000041
edx: c7552732 => 0012ff94
ebx: 05f8884e => 05f88812
esi: 0046745c => 0046745d
```

### •Case 2

```
Stack esp+0 : 45d35066 => 00000001
eax: 00000051 => 0000000a
edx: 00122778 => 45d35066
ebx: 05f87e1f => 05f87e43
esi: 0046dee0 => 0046dee1
```

```
0041743c: push    dword ptr [esp]
00411521: add     esp, 4
00418157: sub     esp, 4
0041743f: mov     edx, [esp]
004164a7: mov     [esp], edi
004164aa: push    492Fh
0041656f: mov     [esp], esp
00416572: add     dword ptr [esp], 4
00416576: pop     edi
00416577: add     edi, 4
0041657d: push    ecx
0041657e: mov     ecx, 4
00416583: push    ebp
00416584: mov     ebp, 6CCD4616h
0041ae2d: xor     ebp, 57B91AC5h
00418ee5: sub     ebp, 55F6569Eh
00418eeb: neg     ebp
00418eed: add     ebp, 0C922DAFh
00418ef3: add     edi, ebp
00418ef5: pop     ebp
00414b58: add     edi, ecx
00414b5a: sub     edi, 2714277Ah
00414b60: push    dword ptr [esp]
00414b64: add     esp, 8
00414b6a: xchg    edi, [esp]
00414b6d: pop     esp
```

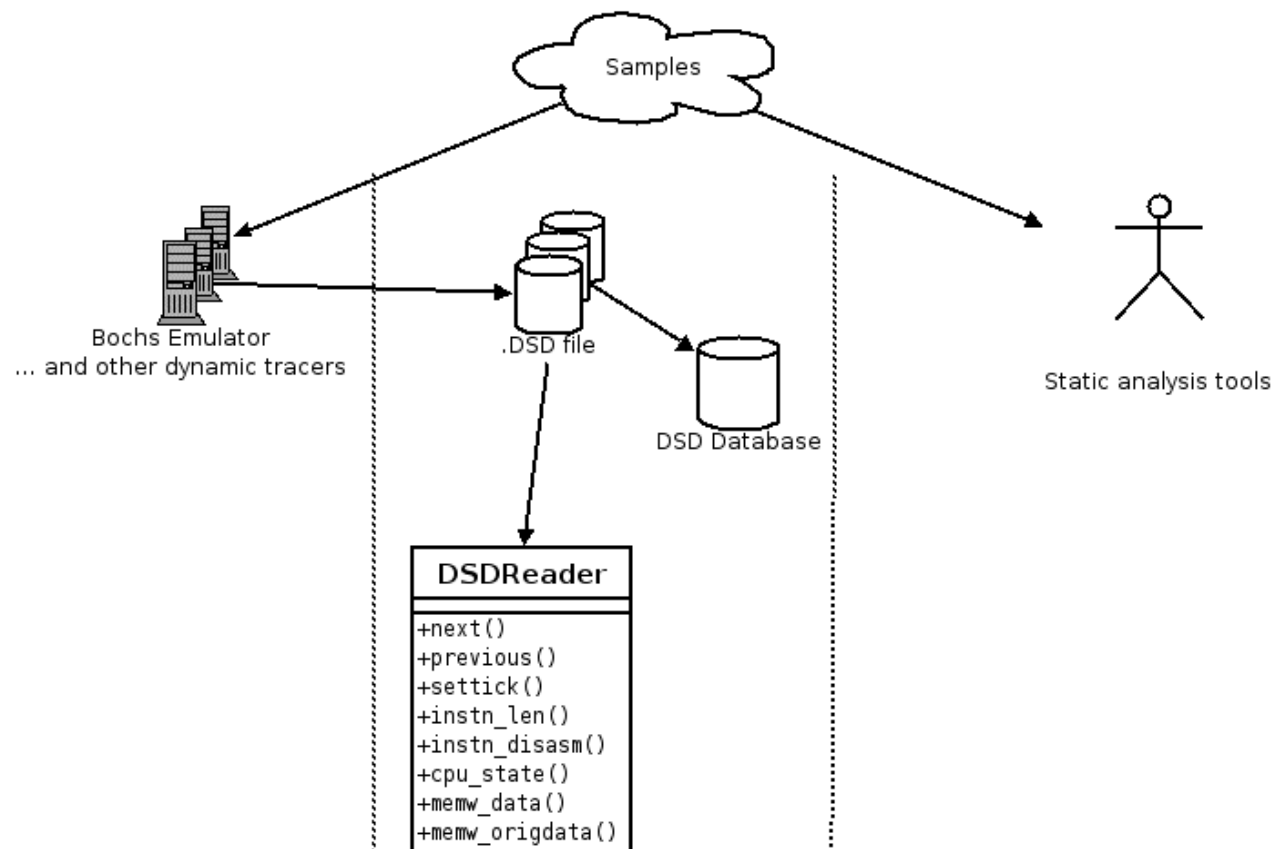
## How to understand VM

- Difficult to manually analyze each handler
  - Huge amount of information
  - Non-reusable analysis
- Need for Automation

0041743c: push dword ptr [esp]  
 00411521: add esp, 4  
 00418157: sub esp, 4  
 0041158f: mov dword ptr [esp], edi  
 004164a7: mov [esp], edi  
 004164aa: push 492Fh  
 0041656f: mov [esp], esp  
 00416572: add dword ptr [esp], 4  
 00416576: pop edi  
 00416577: add edi, 4  
 0041657d: push ecx  
 0041657e: mov ecx, 4  
 00416583: push ebp  
 00416584: mov ebp, 6CCD4616h  
 0041ae2d: xor ebp, 57B91AC5h  
 00418ee5: sub ebp, 55F6569Eh  
 00418eeb: neg ebp  
 00418eed: add ebp, 0C922DAFh  
 00418ef3: add edi, ebp  
 00418ef5: pop ebp  
 00414b58: add edi, ecx  
 00414b5a: sub edi, 2714277Ah  
 00414b60: push dword ptr [esp]  
 00414b64: add esp, 8  
 00414b6a: xchg edi, [esp]  
 00414b6d: pop esp

X 100+

# Automatic analysis tool - DSD-Tracer



(1) Boris Lau, DSD-Tracer: Experimentation and Implementation, Virus Bulletin 2007

# VMProtect

The introductory virtualization obfuscator

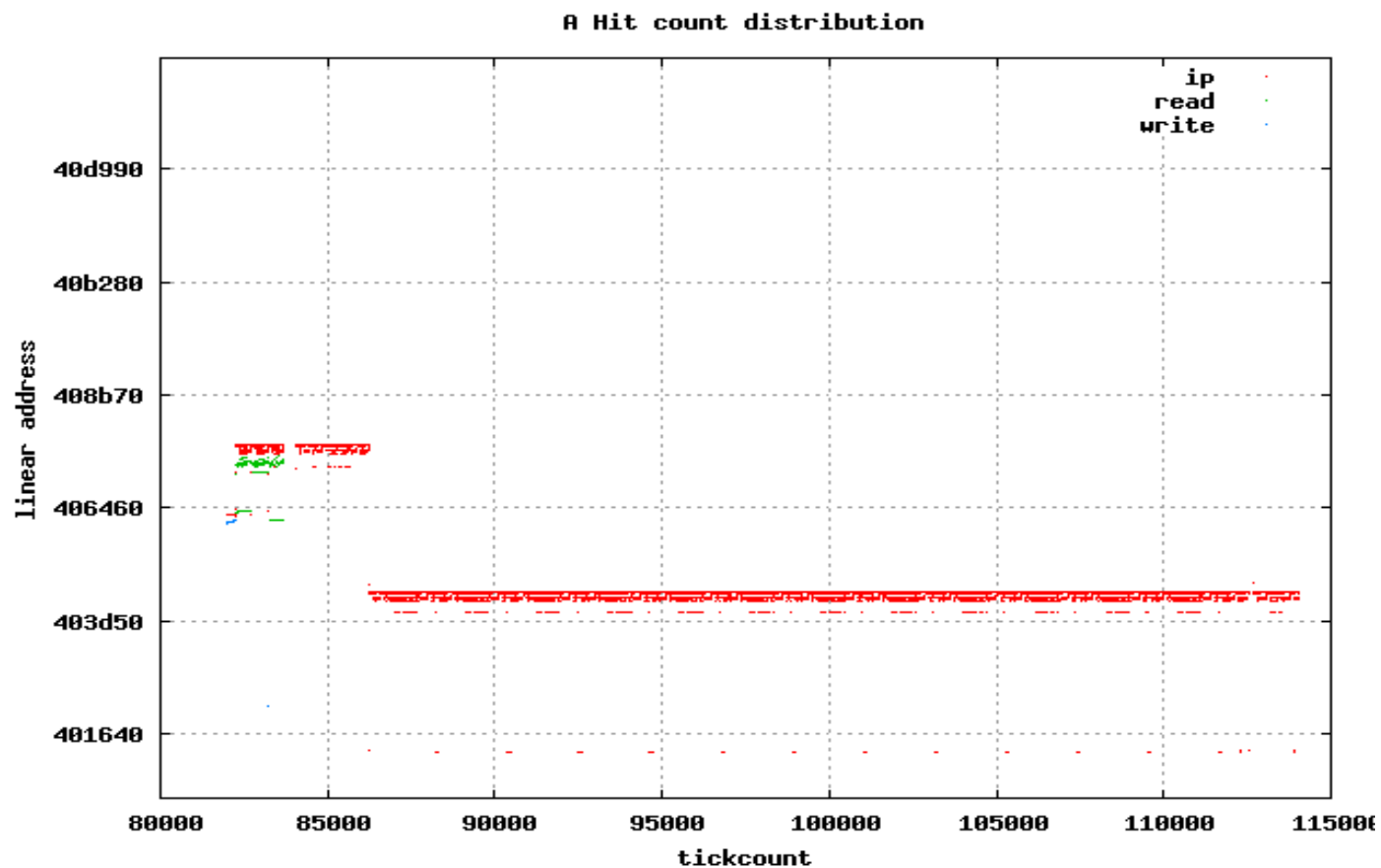
Introduction

Analysis case studies

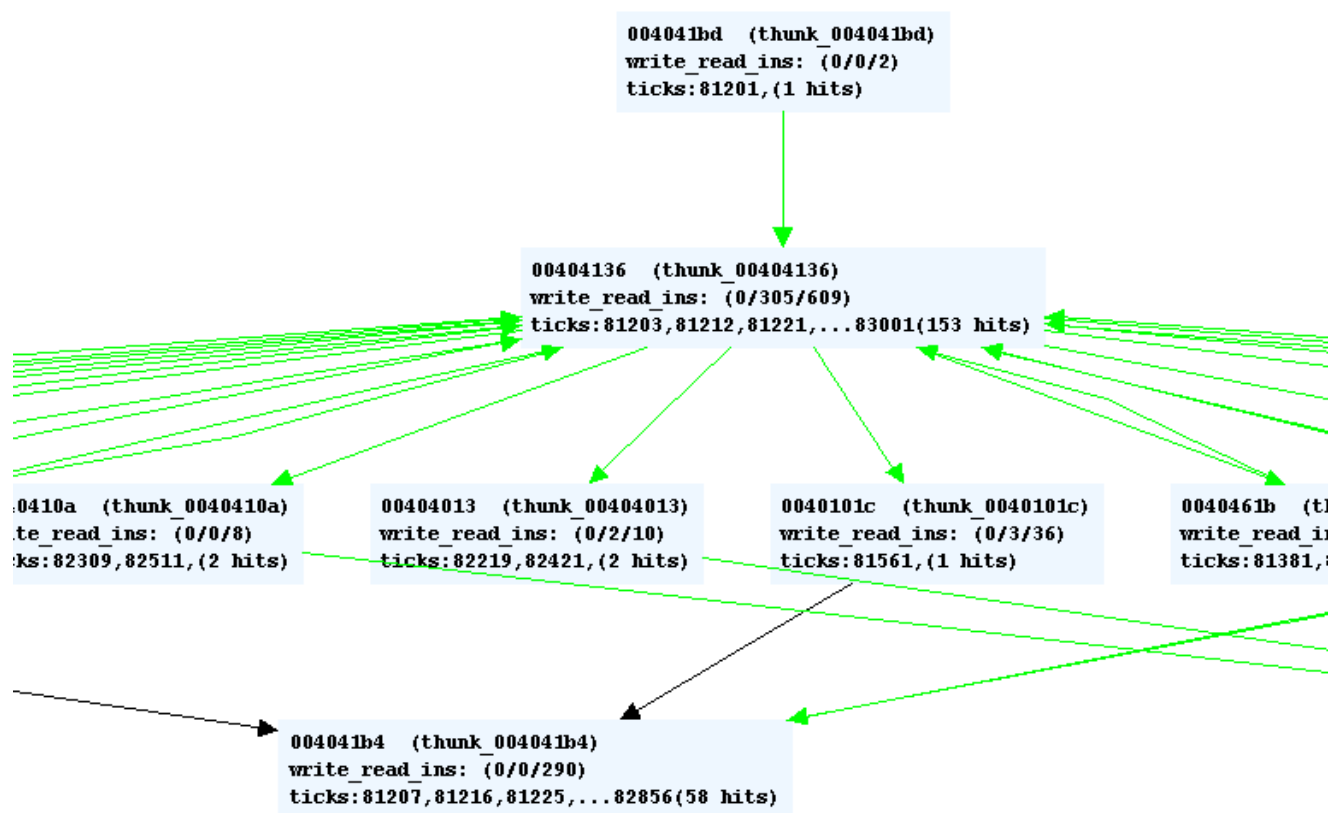
Designing Detection

Technicality with detection

## VMProtect



## VMProtect – control flow



## VMProtect – control flow

- Randomize Opcode with each VM
- Relatively clean code: e.g.

```
and      al, 111100b
mov      edx, [edi+eax]
sub      ebp, 4
mov      [ebp+0], edx
jmp      CheckVMStackOverflow
```

- EDI points to the context (16 internal registers)
- EBP used as an internal stack

# Themida

Introduction

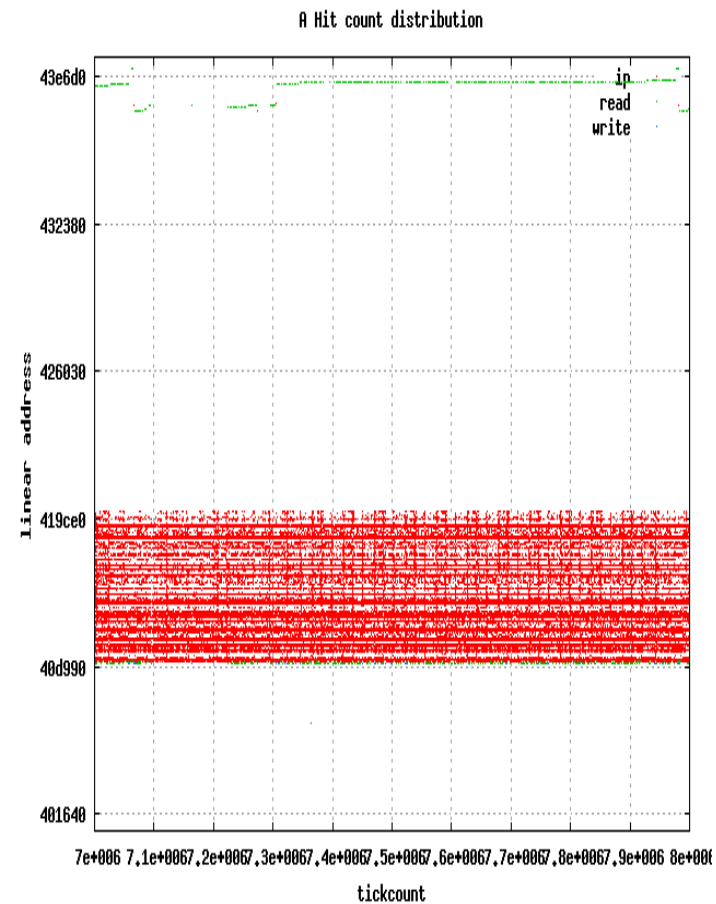
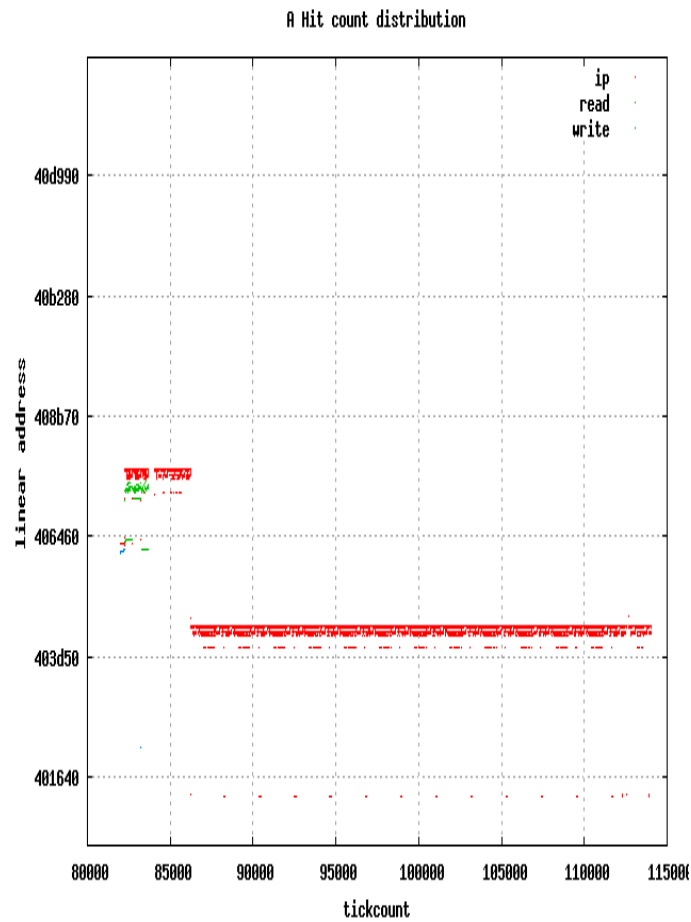
Analysis case studies

Designing Detection

Most commonly used virtualization obfuscator

Technicality with detection

# Themida vs. VMProtect

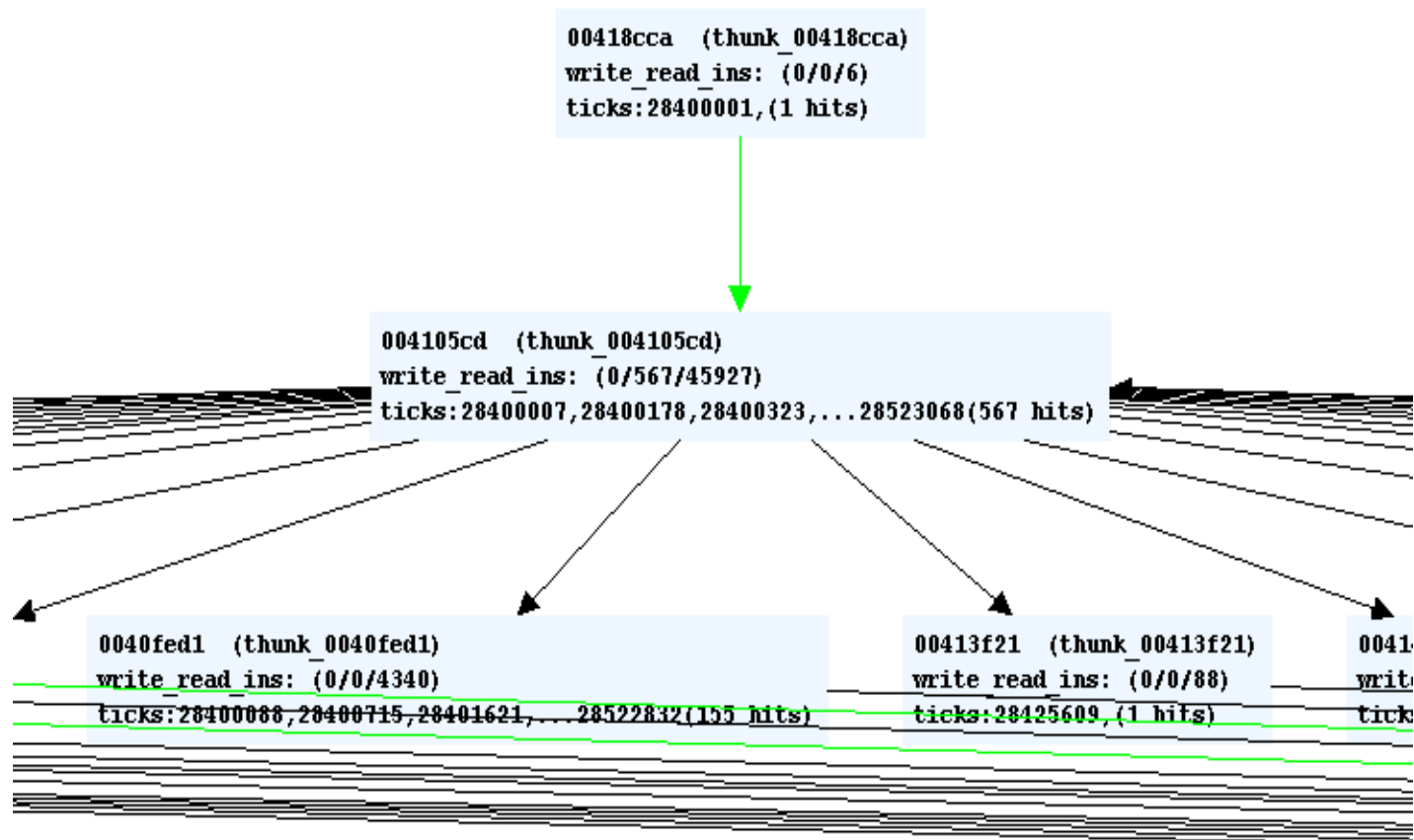


## Themida vs. VMProtect

	Number of virtualized Bytecode	Number of instructions per Bytecode	Estimate number of instructions handled
Goat file	-	-	12
VMProtect	162	20	3240
Themida	258	200	51600

(1) Figure exclude instruction required to decrypt the sample

# Themida



## Themida – Metamorphic VM

- EDI points to context as well as the VTable
- Opcodes are encrypted with unique bytes before indexing into VTable
- Heavily metamorphic
  - Randomly inserted jumps
  - Constants required are generated via arithmetic instructions
  - Register morphing by using top of stack as swapping
  - Known condition jumps

```

push    dword ptr [esp]
mov     ecx, [esp]
jmp     loc_4174FC
push    dword ptr [esp]
mov     ecx, [esp]
jmp     loc_4174FC
push    6AFAn
mov     [esp], edx
jmp     loc_40FB05
mov     edx, esp
push    ebp
mov     ebp, 3FD3296Ah
jmp     loc_41960F
sub     ebp, 3FD32966h
add     edx, ebp
pop     ebp
sub     esp, 4
mov     [esp], ebx
push    edx
mov     edx, 4
mov     ebx, edx
mov     edx, [esp]
add     esp, 4
jmp     loc_4132DB
jmp     loc_41AC1C

```

# ExeCryptor

VM with hidden dispatcher

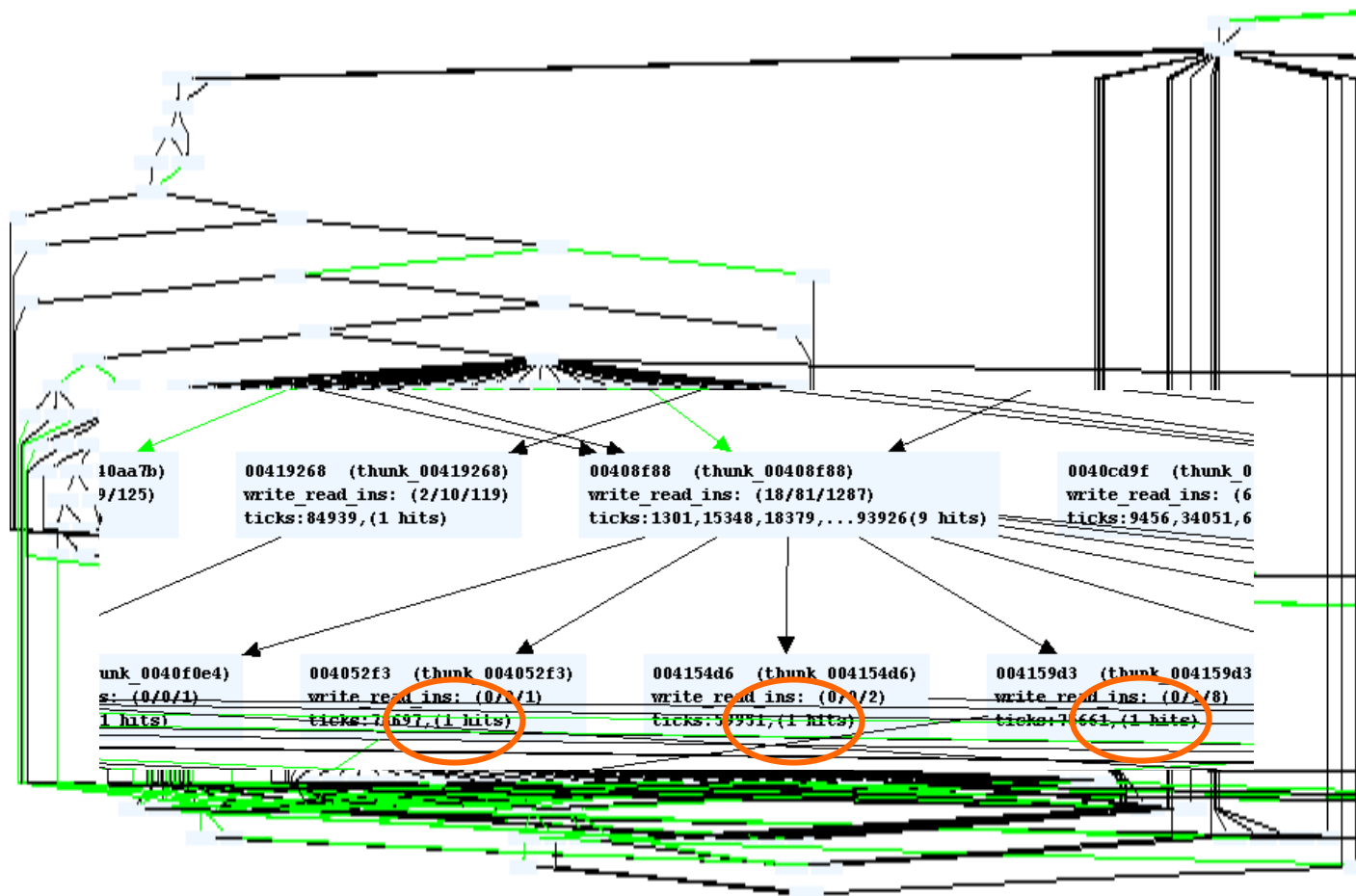
Introduction

Analysis case studies

Designing Detection

Technicality with detection

## ExeCryptor



## ExeCryptor – purple pill technique



```

    Jmp ins_before_bytecode
ins_before_bytecode:
    <junk_ins>
    Call VMEnter
    <bytecode>
VMExit:
    <junk_ins>
    Jmp <next bytecode>

```

- Control flow of the bytecode depends on native instructions
  - Use the [esp+4] as the pointer to the bytecode
- Constantly jumping in and out of virtualization

# Detection strategy

What can we do about the information?

Introduction

Analysis case studies

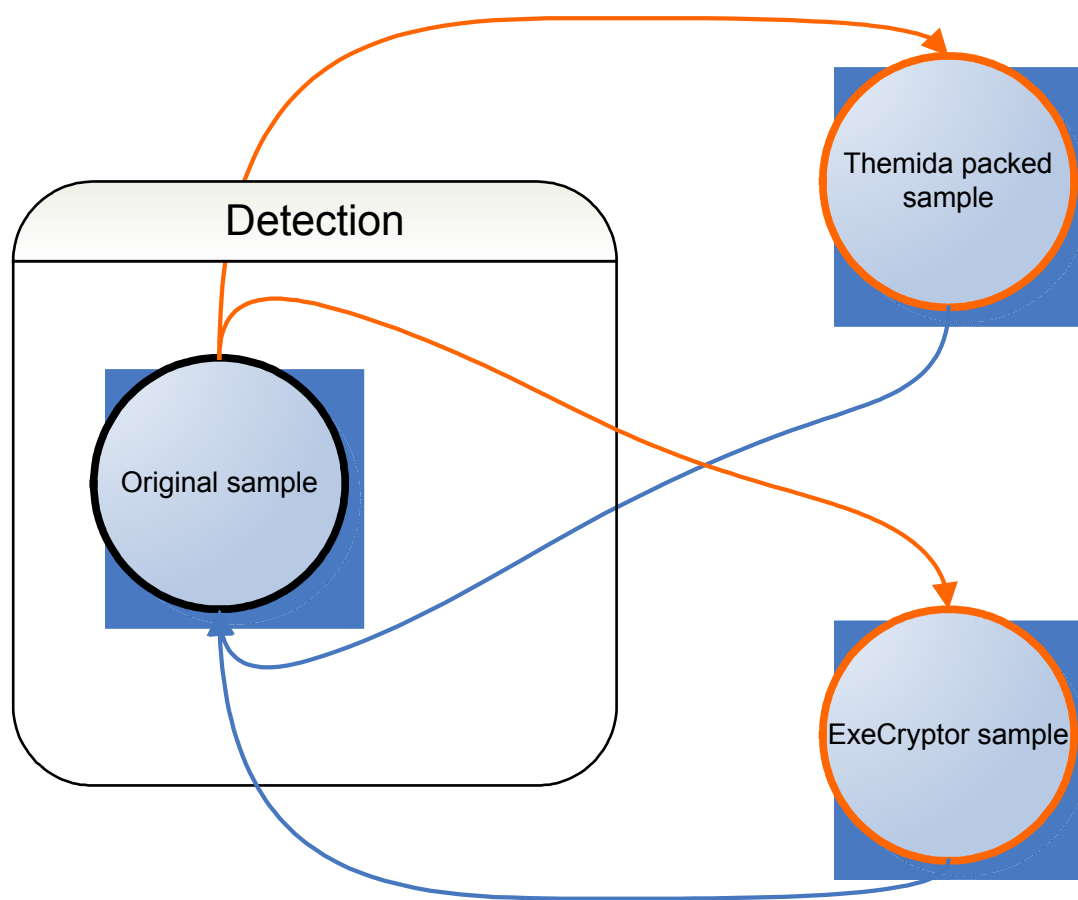
**Designing Detection**

Technicality with detection

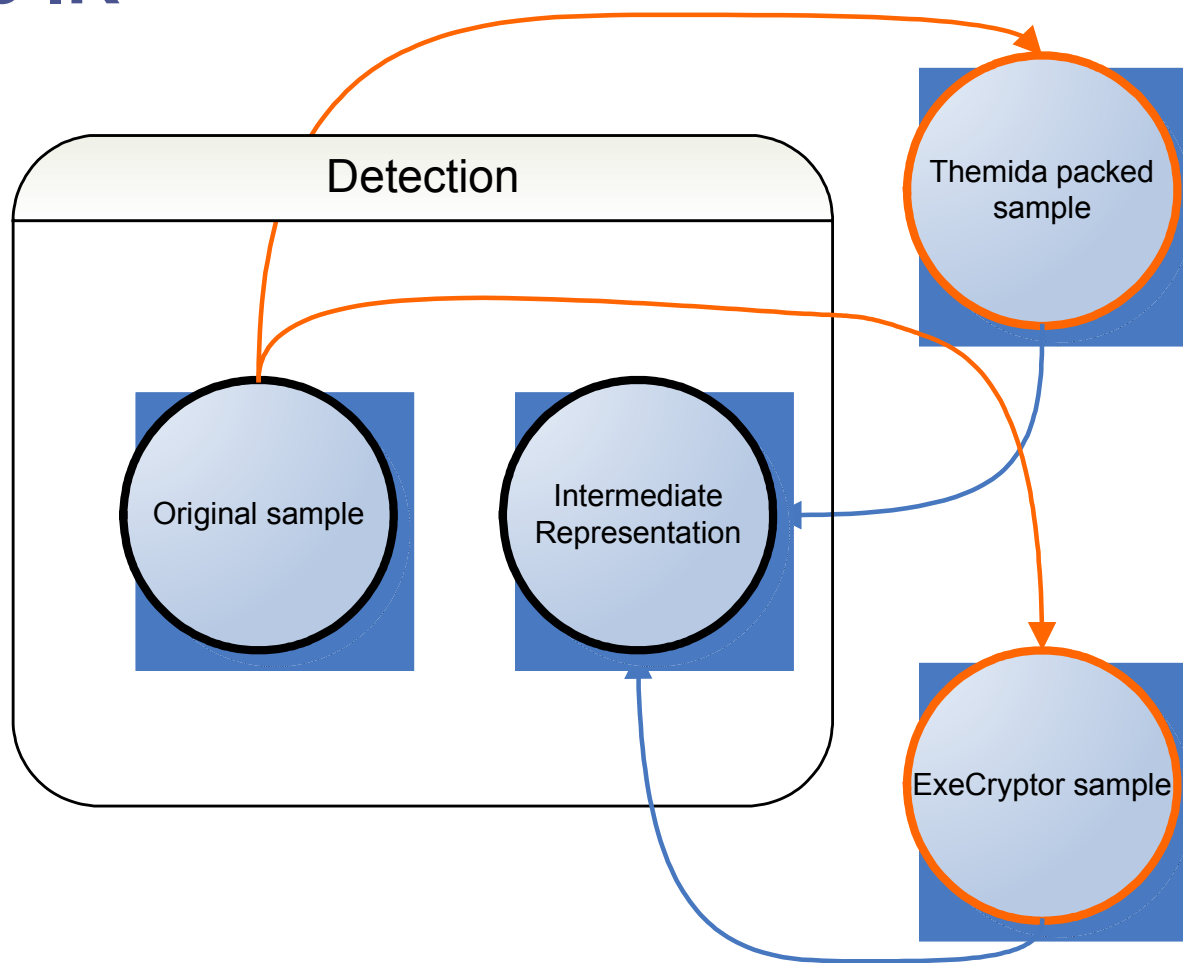
## Overview of strategy

- Extract information from the Virtualization sample
  - Junk obfuscation in the Virtual machine
  - The morphing of bytecode makes it very difficult to make prior assumptions about the Virtual Machines
- Detection
  - How do we unify information extracted from various Virtual Machine such that it can be used in detection?

## Possible strategy 1 – decompiling into original code



## Possible strategy 2 – translating into IR



## Intermediate Representation (IR)

- Existing research on abstraction techniques for detection
  - Semantic dependency <sup>(1)</sup>
  - Control flow <sup>(2)(3)</sup>
  - System call <sup>(4)</sup>
  - Binary data referenced
- All these information can be obtained via emulation
  - See relevant research

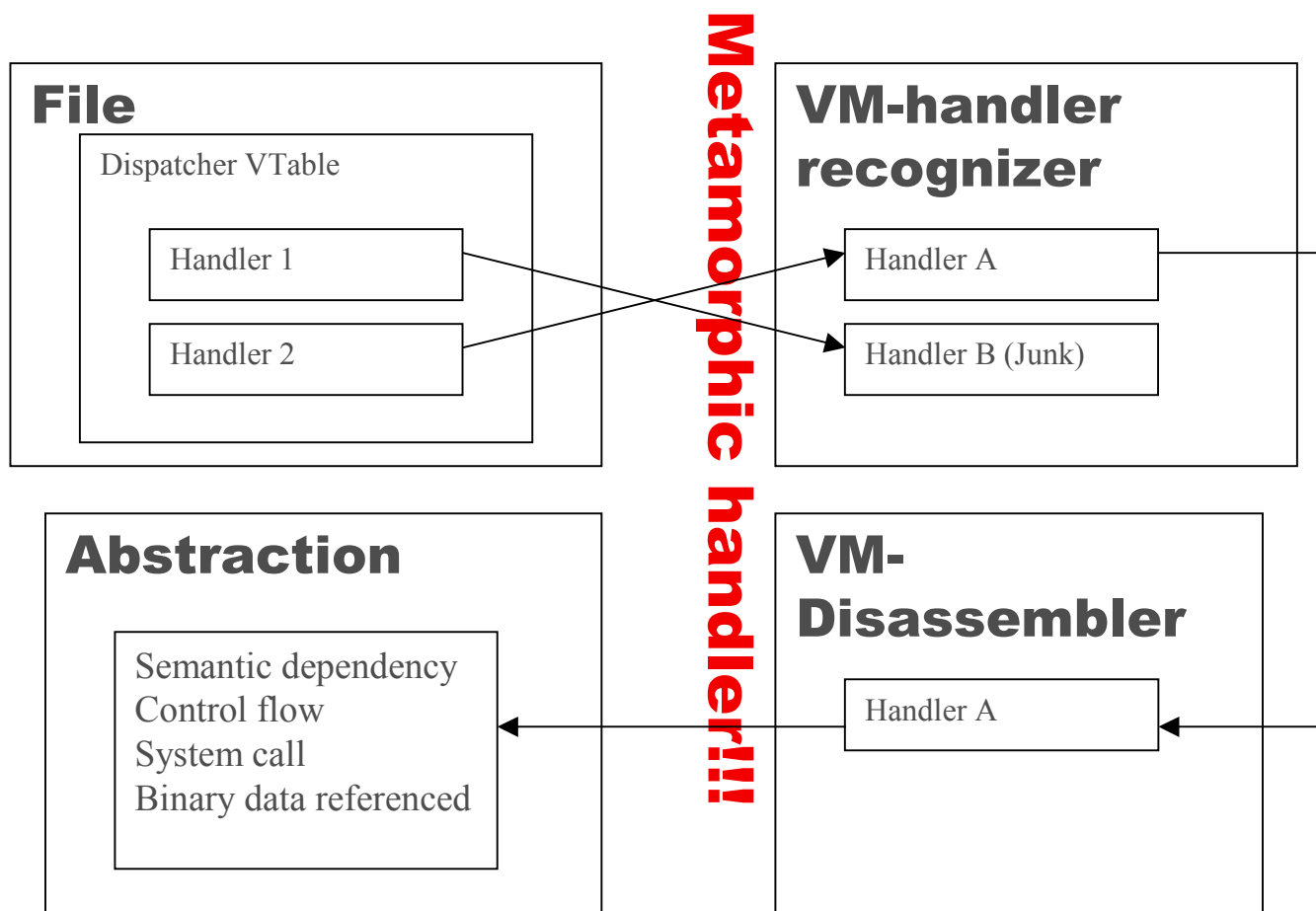
(1) Christodorescu et al. , Semantic aware - <http://www.eecs.berkeley.edu/~sseshia/pubdir/oakland05.pdf>

(2) Lo et al. R.W. Lo, K.N. Levitt, and R.A. Olsson. Mcf: Malicious code filter.

(3) control flow analysis, Digital genome mapping - [http://www.f-secure.com/weblog/archives/carrera\\_erdelyi\\_VB2004.pdf](http://www.f-secure.com/weblog/archives/carrera_erdelyi_VB2004.pdf)

(4) A. Mori, T. Izumida, T. Sawada, and T. Inoue. A tool for analyzing and detecting malicious mobile code.

## Partial disassemble



## Automatic extraction of information about handler

- Emulation
  - As with Metamorphic viruses
- We emulate through the handler and see what part of context had been affected
- Works across different VM
  - need to know starting point/ending point of handler

# Extracting the information

Introduction

Analysis case studies

Designing Detection

A low level approach to efficient detection

Technicality with detection

## Problems with emulation

- Emulation through the handler is inefficient
  - Themida could require as many as 5000 times the number of original instructions
  - And this is not considering the emulation overhead
- Increase volume of instructions cause big performance impact for emulation
- We need some way of making things more efficient
  - We propose using Dynamic Binary Translation (DBT)

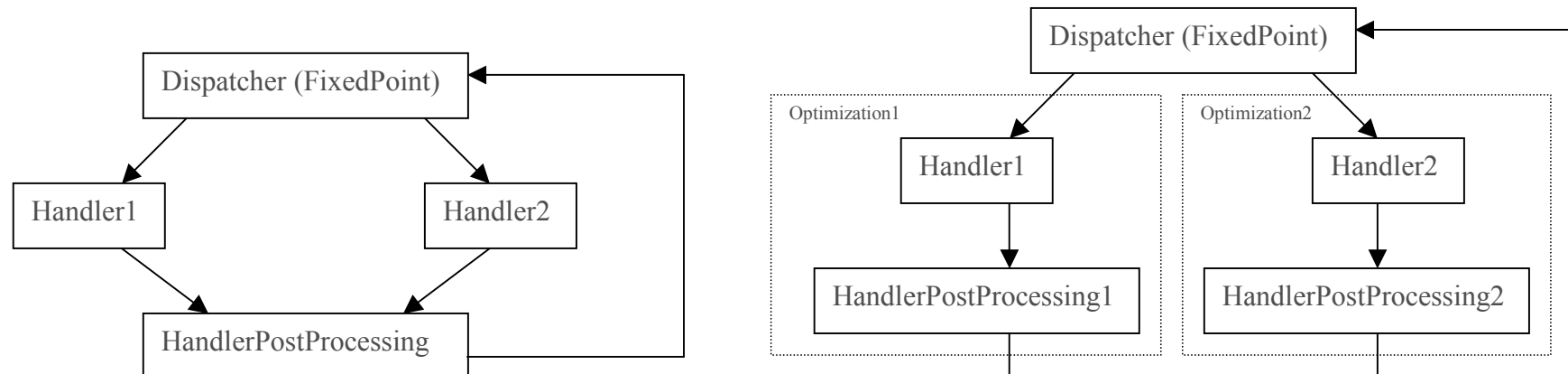
## Dynamic binary translation (DBT)

- Instructions are translated at runtime to the native instructions operating on the emulated state.<sup>(1)</sup>
  - If the same instruction is encountered again, the translated instructions will be executed instead.
- VM is perfect for DBT!
  - Handlers are executed multiple times
  - Can combine DBT parse with the decision processing of which bytecode to disassemble
- Borrow know compiler optimization techniques
  - Constructing use-def chain via Static Single Assignment (SSA)<sup>(2)</sup>

(1) Full potential of dynamic binary translation for AV emulation engine, Jim Wu Internet Security Systems, Virus Bulletin 2006

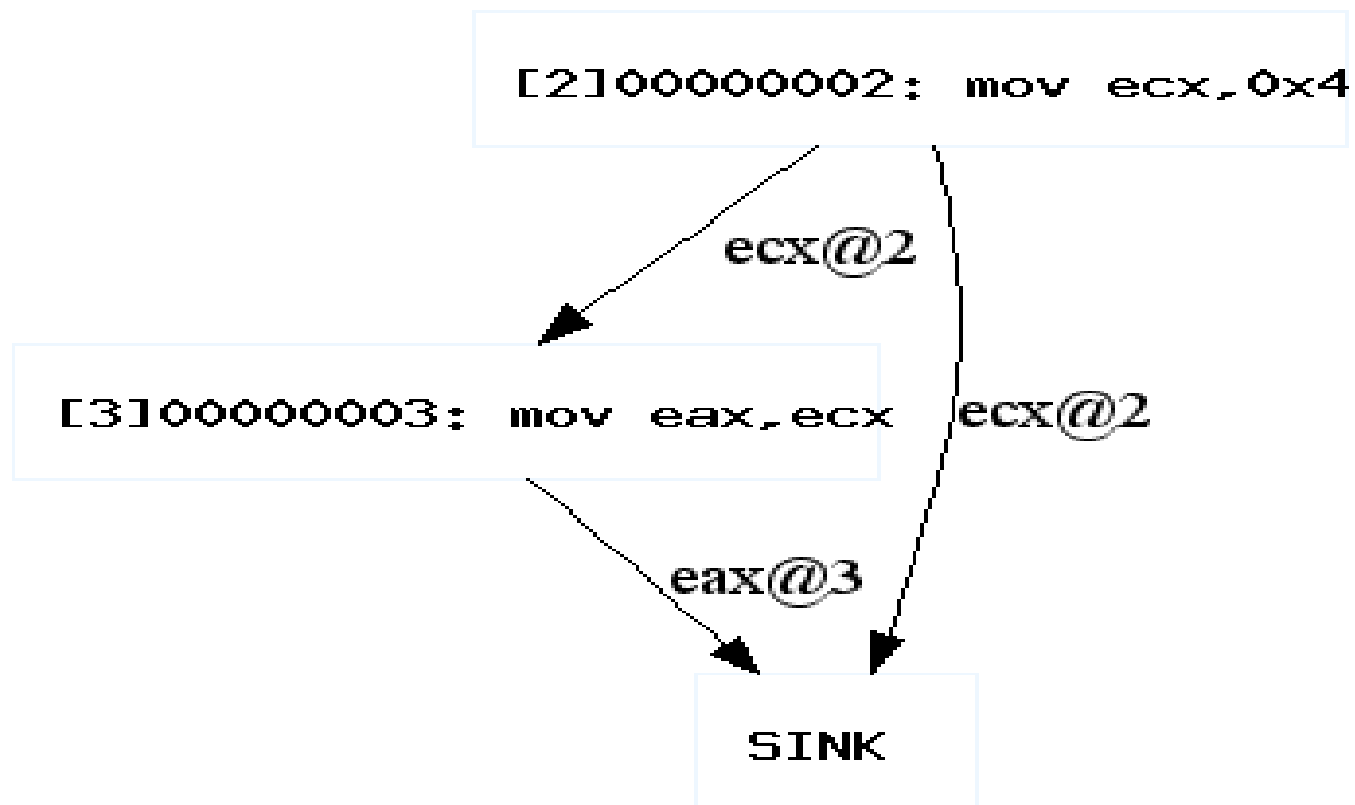
(2) [http://en.wikipedia.org/wiki/Static\\_single\\_assignment\\_form](http://en.wikipedia.org/wiki/Static_single_assignment_form)

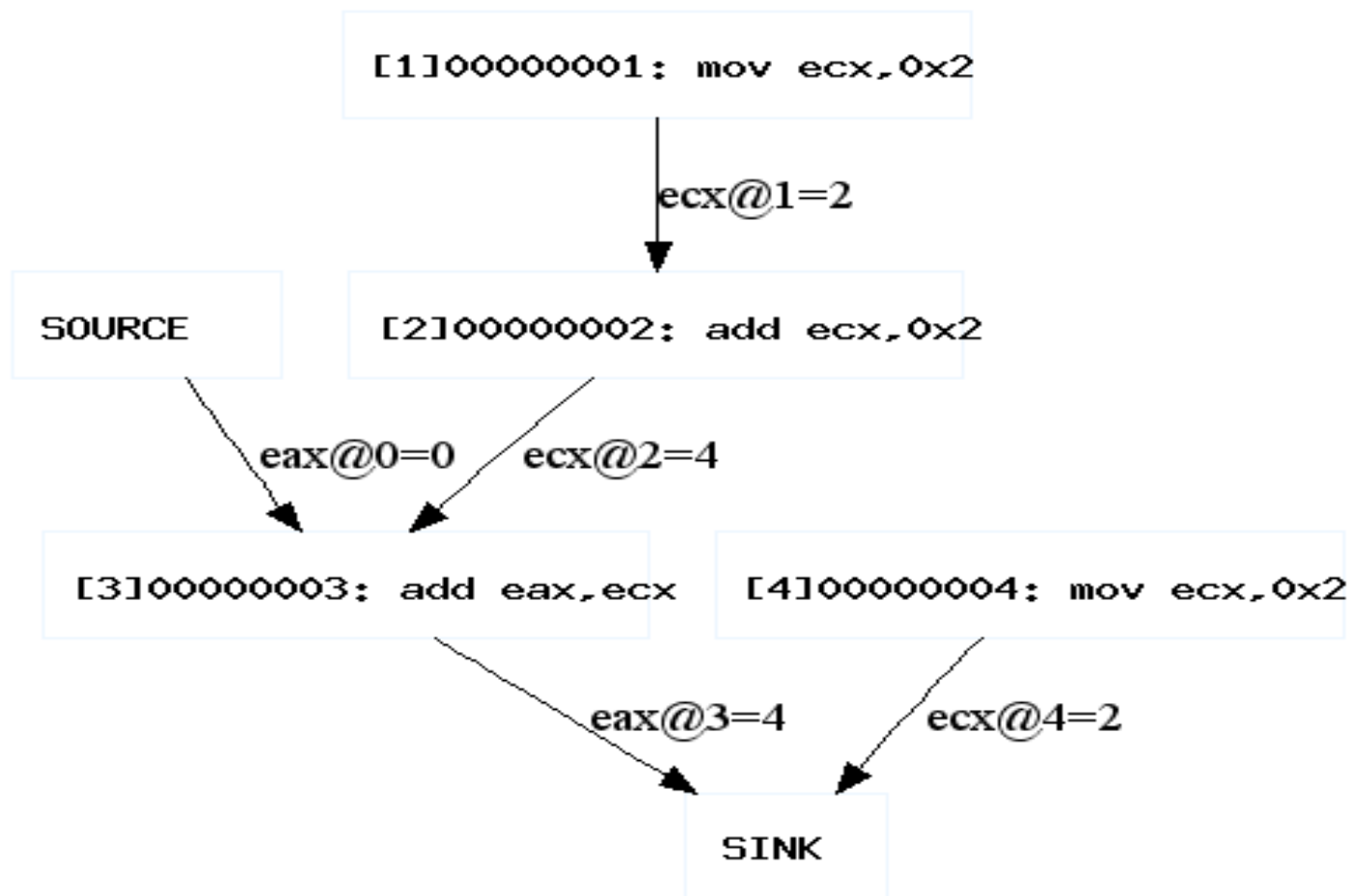
## Local DBT on handler

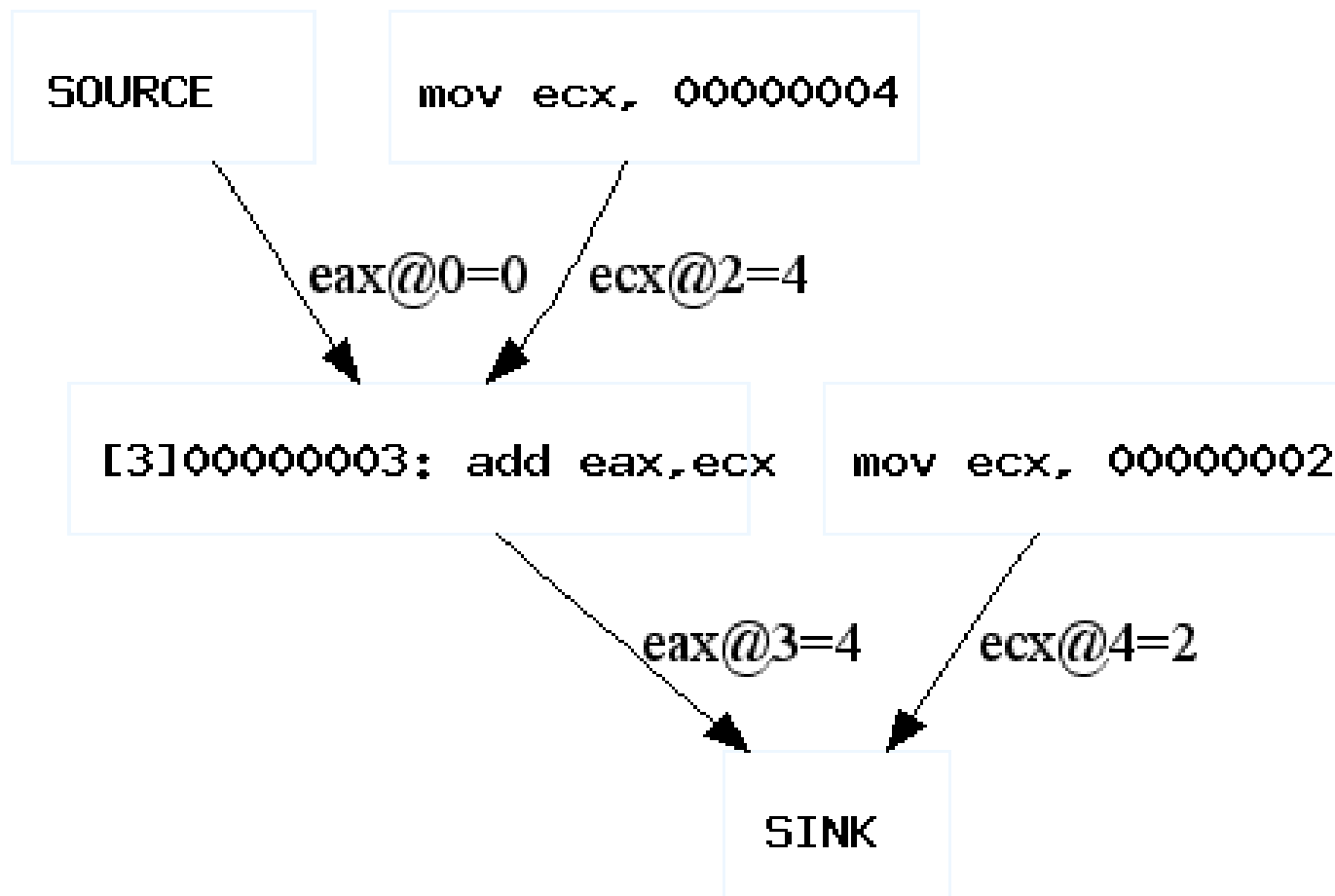


- DBT normally requires global control flow knowledge
- We apply DBT locally on each handler
- Only require one pass per handler – Quicker than traditional DBT

## Constructing use-def chain







SOURCE

[1]000000001: mov eax,0x8

[2]000000002: mov ecx,0x4

[3]000000003: mov eax,ecx

SINK

ecx@2

ecx@2

eax@3

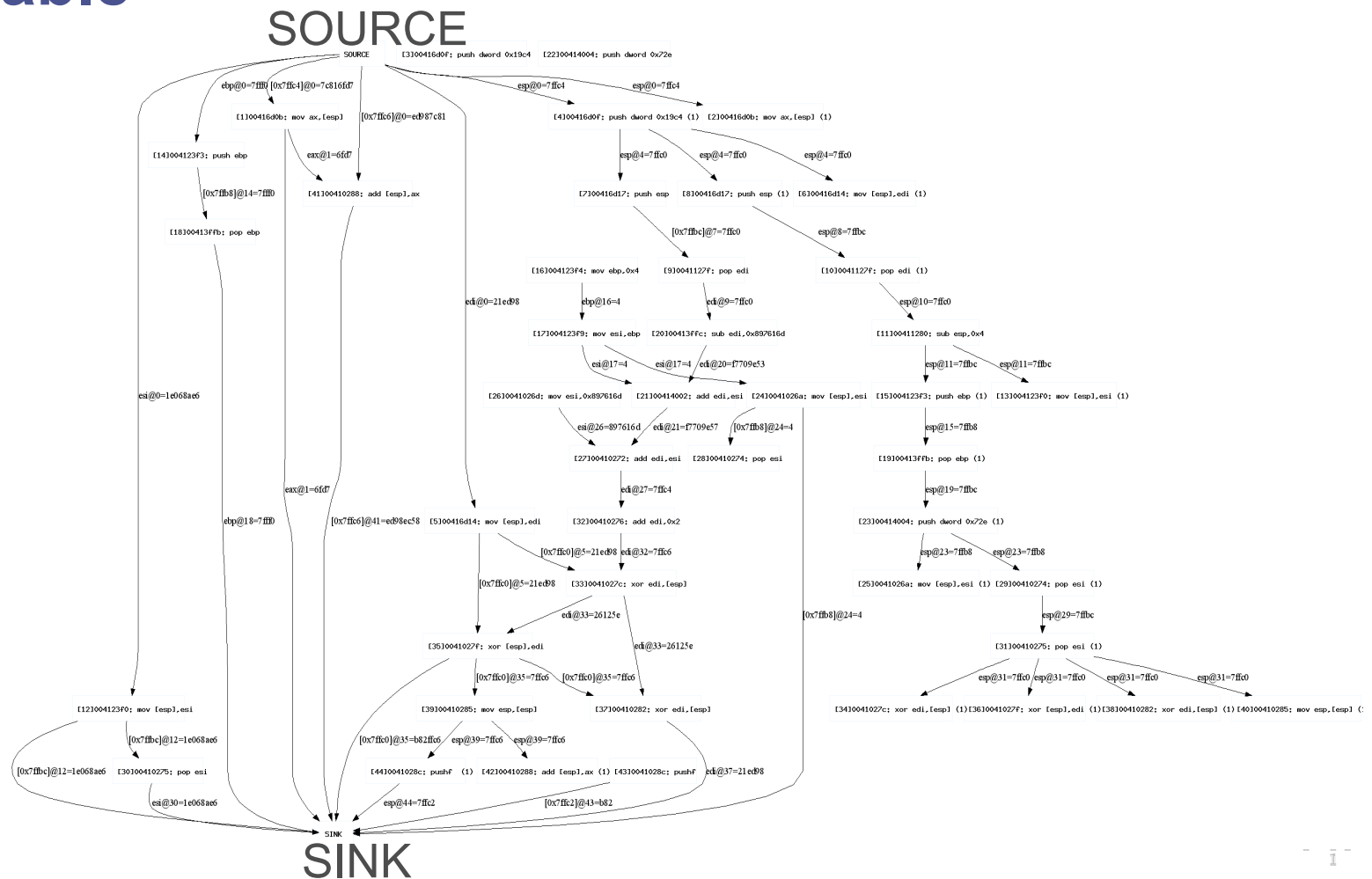
sophoslabs

```

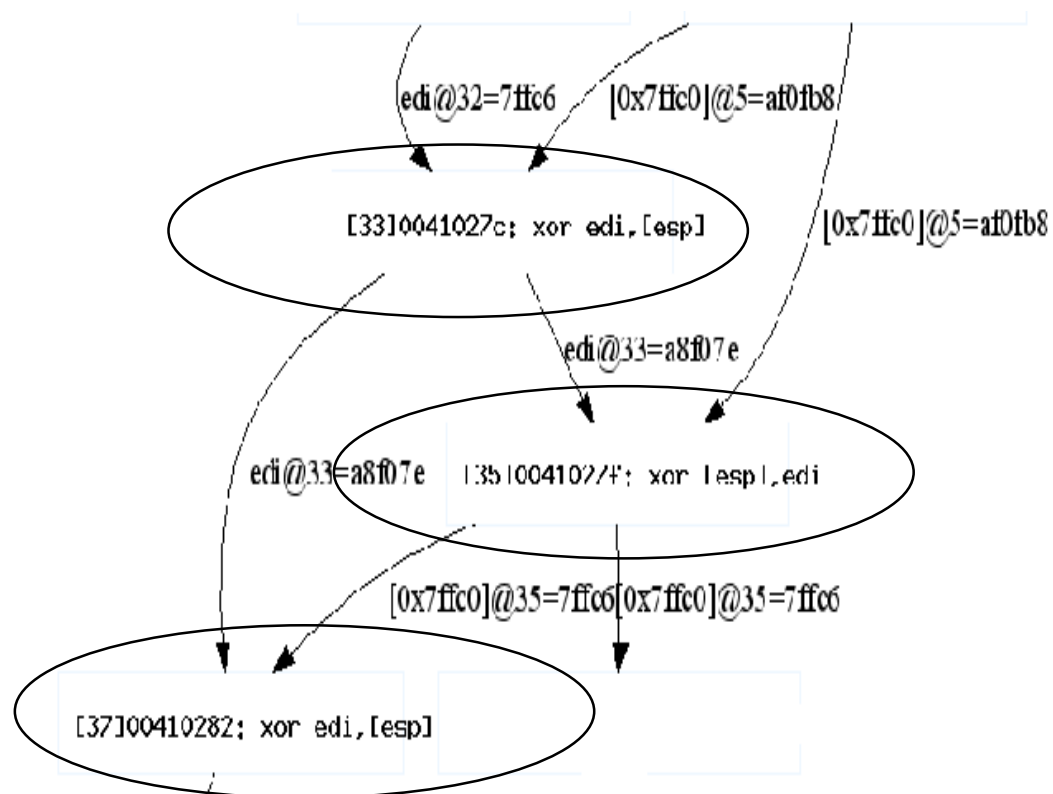
1 1
0101
011111
11000
10001
1 01
11001
01010
01010
10101
1 1
1

```

# Example optimization –unused variable



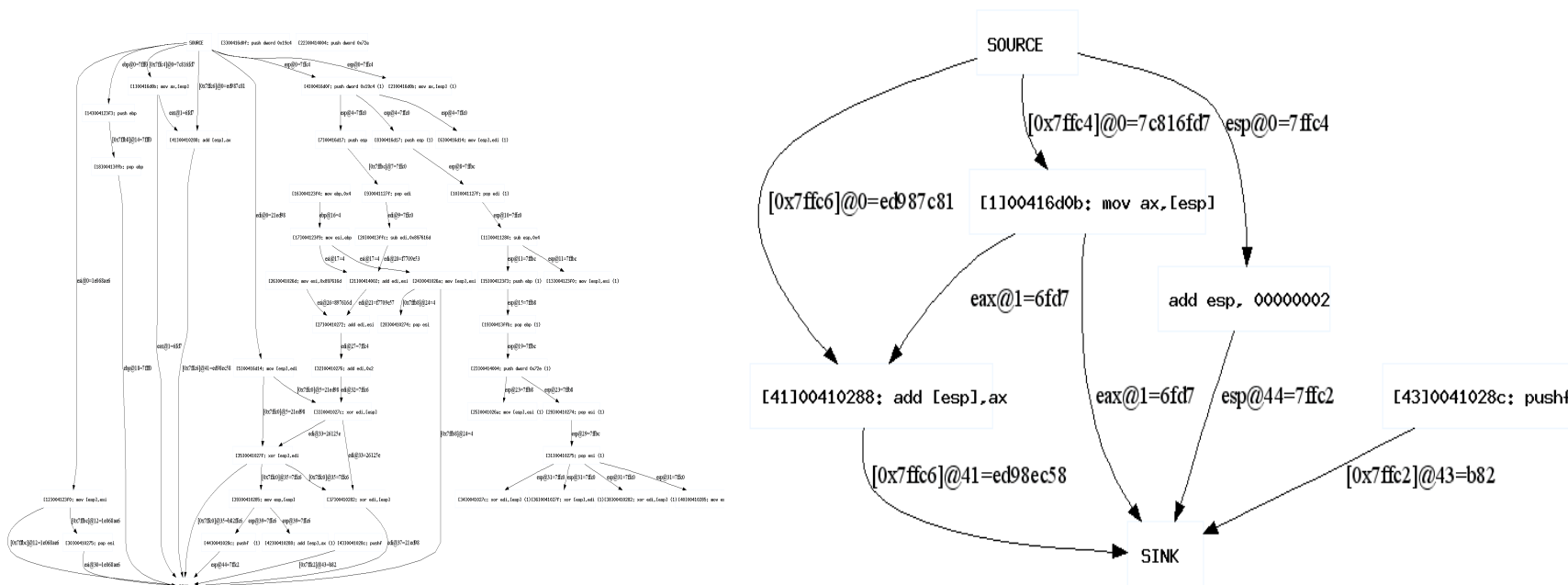
## Example of obfuscation used by Themida



```

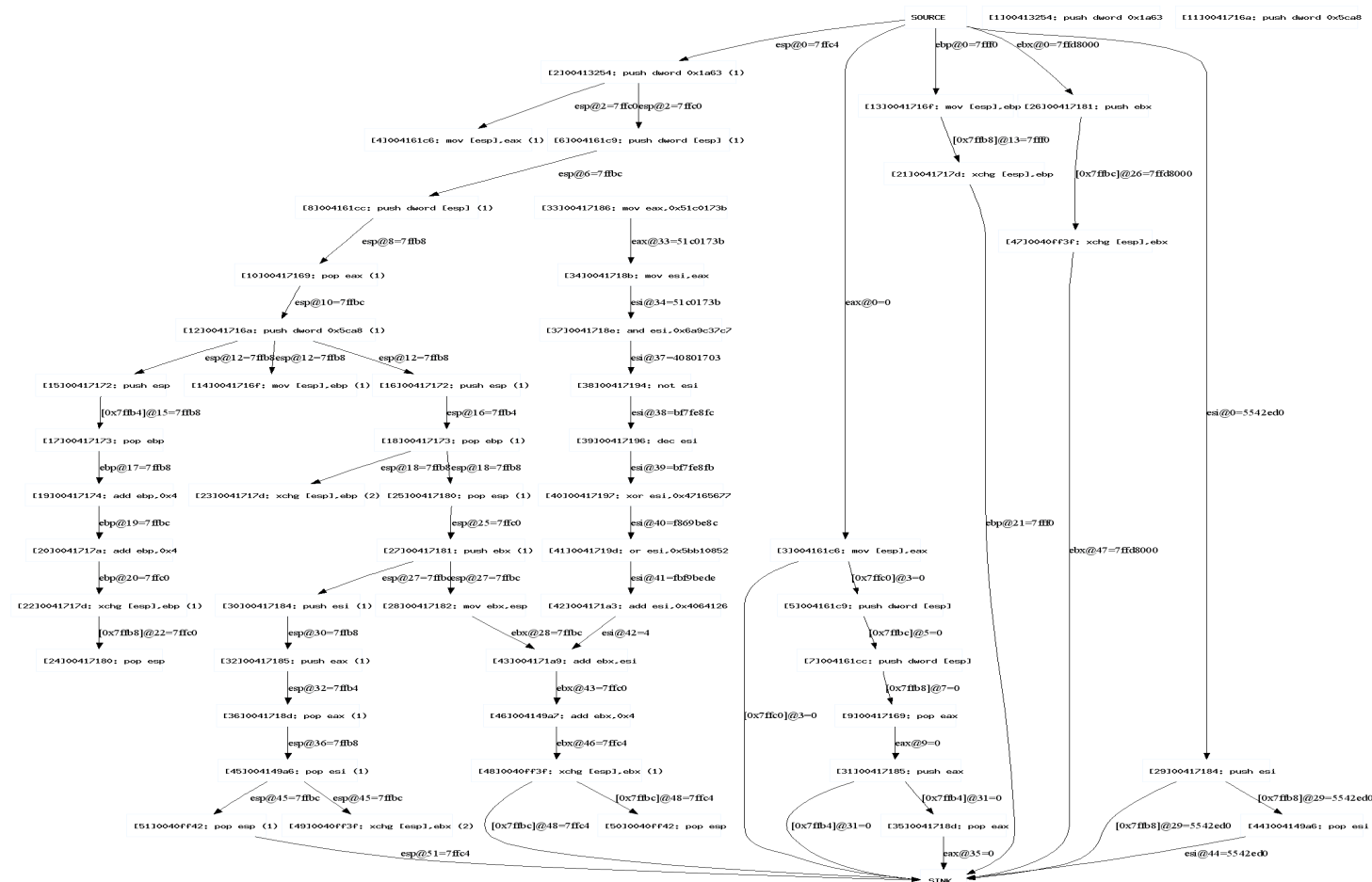
edi@37
= edi@33 ^ [esp]@35
= (edi ^ [esp]) ^
  ([esp] ^ (edi ^ [esp]))
= [esp]
  
```

# Example of Optimization of Themida Handler



- From a 46 node graph to 6 node graph (767% performance)
- Also provide insight into what the handler would do

## Some example use-def graphs



## Some example use-def graphs

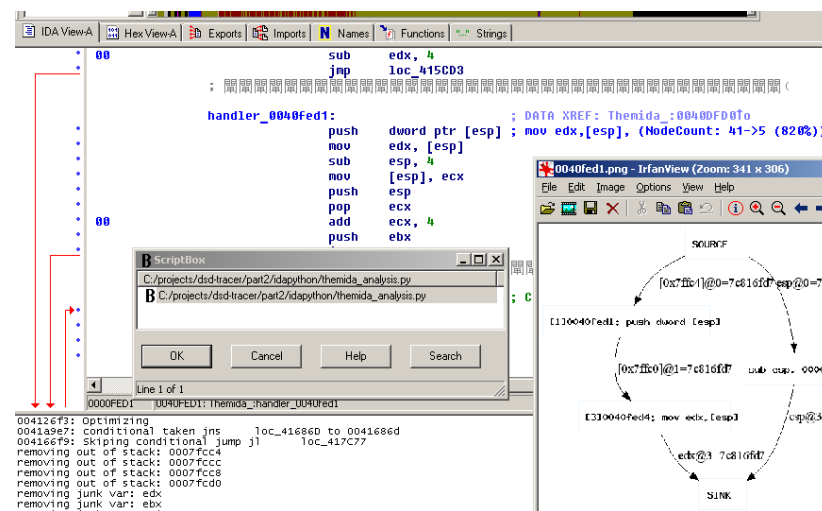


SINK

A nop instruction (53 nodes)

# POC Implementation

- Implemented in Python
  - Pydasm, Pydot – Ero Carrera
  - Pgraph, PyDbg – Pedram Amini
- Can be integrated with emulator
  - PyDbg – as a per-instruction emulator
  - DSD-Tracer – playing back a trace
- Integrated with IDAPython
  - Can generate optimization graph from IDA



## Conclusion

Almost the end....



## Summary

- Observe techniques used by various common virtualization obfuscators
- We illustrate techniques for scanning through virtualization obfuscators
  - based on combination of existing tools (Abstraction, Emulation)
  - and some new techniques (Partial disassembly, Handler local DBT)
- “Using virtualization against virtualization”

## “Do I still believe in defeating virtualization obfuscators?”

- Yes
  - the research shows possibility but need improvement
- No
- Virtualization obfuscator developers aim to protect low-level details
- Anti-virus researcher tend to care about high-level abstractions
- No conflict of interest between the 2 parties
  - They don't generate revenue from malware writers

**Thank you**

[boris.lau@sophos.com](mailto:boris.lau@sophos.com)

## Appendix: Is DBT enough?

- About 50 times improvement
  - Instruction DBT provides about 5 times improvement on Themida samples
  - Partial emulation provides another 10 times improvement
- Vs. 5000 times increase in Themida execution time...
  - Factor of 100 to catch up on
- But this is only for virtualization packed samples
  - How deep do you want to emulate?

## Bonus slides: use-def optimization techniques

Low-level dirt about the optimizations

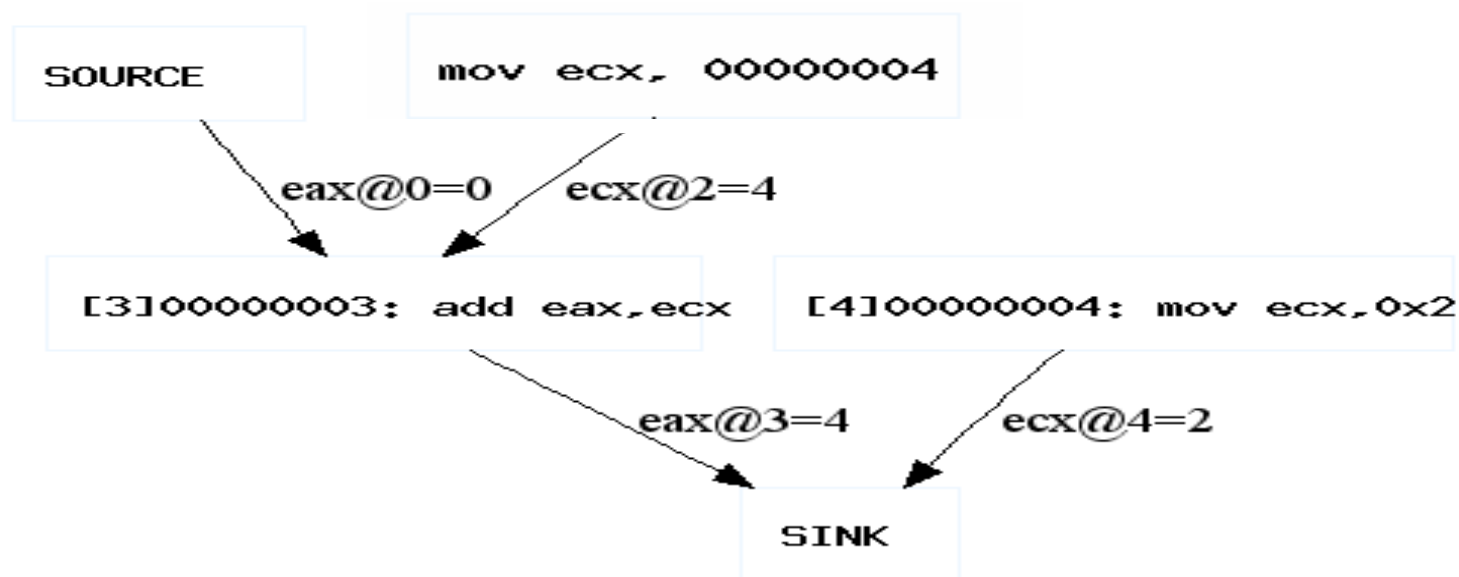


# Optimization techniques

- Remove unused variable
  - Remove nodes which are not in path between source/sink node
- Simplify algebraic operations
  - Resolving constants <sup>(1)</sup>
  - Simplifying arithmetic instructions (add,sub,mul)
- Recognize Context of VM
  - Remove variables outside the VM

(1) Can use PyDbg as the background emulator to resolve constant values for optimization

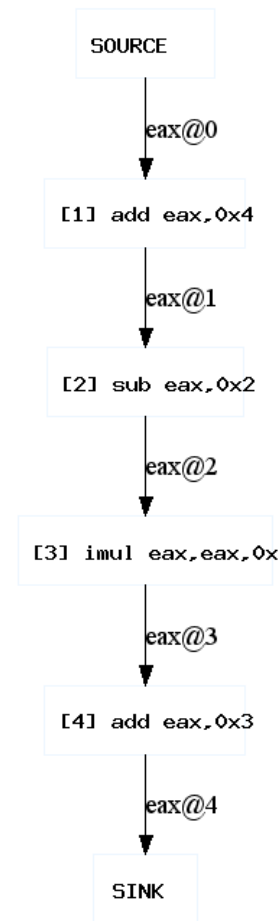
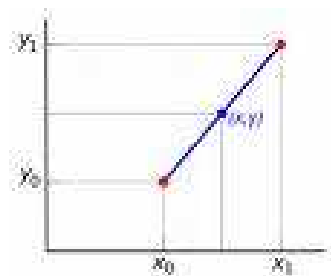
## Optimization 2 – find constant set



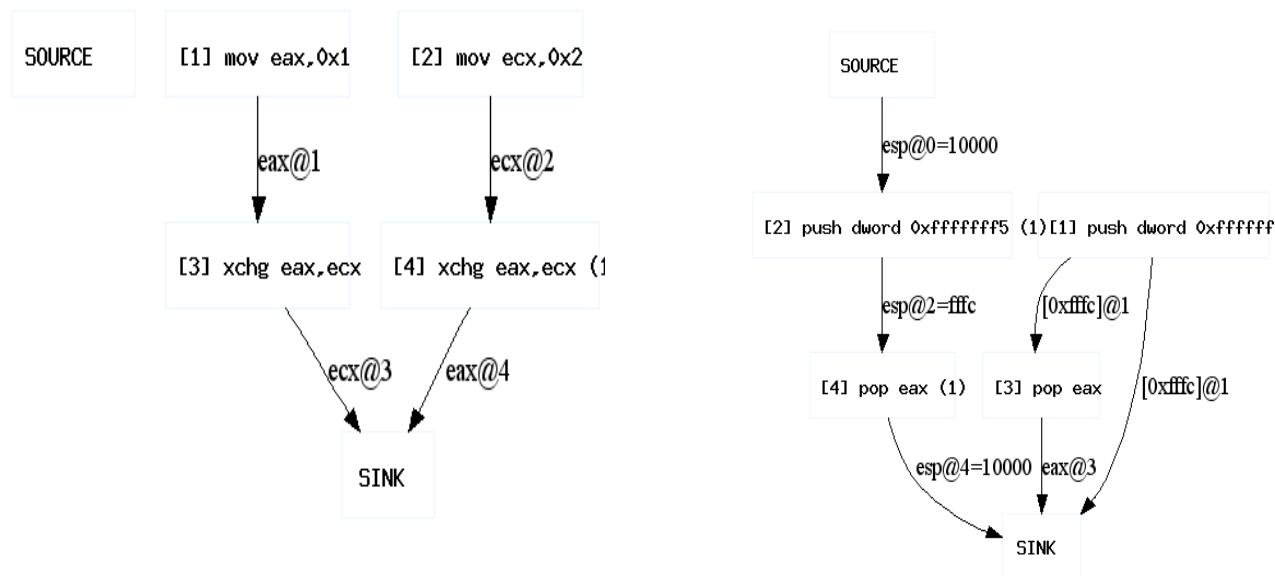
(1) Can use PyDbg as the background emulator to resolve constant values for optimization

## Optimization 2.1 – Simplifying chain of modification

- Seen in Themida where it modifies a variable via a series of simple algebra operations
- Linear interpolation can be used to optimise instruction chain
  - Can be applied to more than one variable



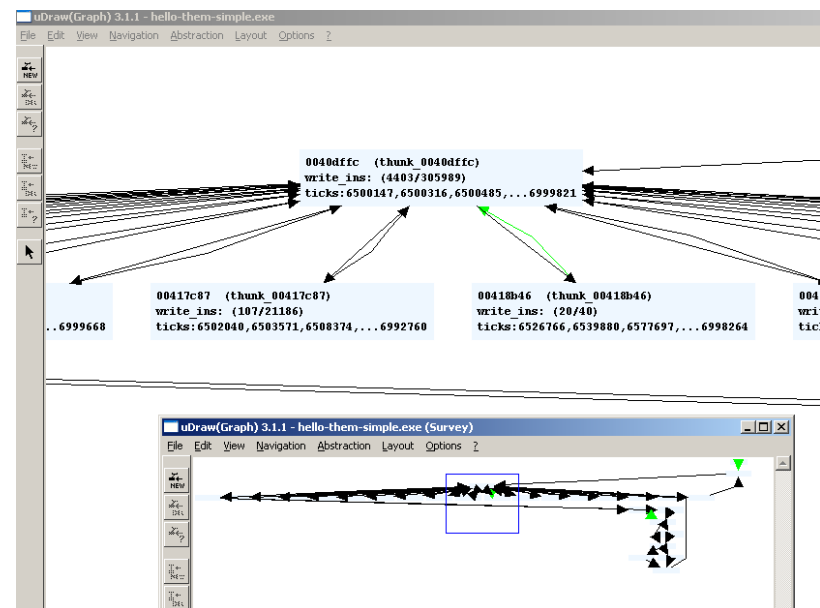
## Detail use-def: Example of splitting use-def chain



Splitting use-def chain within instruction

## Penalty for defactorizing control flow

- If more code is shared between handlers, this will cause huge amount of storage requirement for IR as well as waste of processing by reoptimizing repeated code.
- We can see from the survey view that only 2 of handlers share the same code.



# **Bonus slides:**

## **Techniques for investigating VM**

Introduction

Techniques for analysis

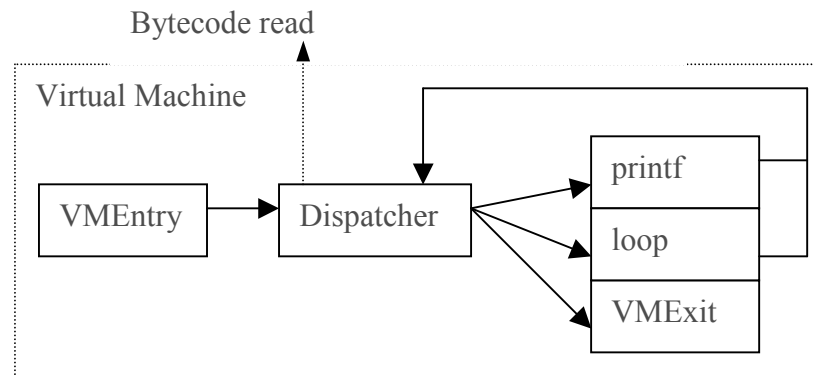
Analysis case studies

Tips and tricks on dealing with virtualization

Designing Detection

Technicality with detection

## Fixed point analysis of the VM



- Virtual machine tends to have some looping structure within its architecture
- We can compare snapshots of the emulation at specific fixed points
- It can be very useful in establishing the VM context

## Bonus slides: Extra case studies about virtualization obfuscators

More about ExeCryptor and Themida API virtualization



## VMProtect – VMEntry and VMExit

```

00404117 StartOfByteCode = dword ptr 18h
00404117
00404117      push    eax                ; save the registeres
00404118      push    esi
00404119      push    ebp
0040411A      push    ebx
0040411B      push    edx
0040411C      pushf    |
0040411D      push    ebp
0040411E      push    edi
0040411F      push    ecx
00404120      push    0
00404125      mov     esi, [esp+14h+StartOfByteCode]
00404129      mov     ebp, esp
0040412B      sub     esp, 0C0h          ; allocate the C0 VM context space
00404131      mov     edi, esp          ; edi points to the base of context
00404132
0040101C      mov     esp, ebp
0040101E      pop     edx
0040101F      pop     ecx
00401020      pop     edi
00401021      pop     ebp
00401022      popf
00401023      pop     edx
00401024      pop     ebx
00401025      pop     eax
00401026      pop     esi
00401027      pop     eax
00401028      retn

```

## Processor Specifications



Processor Type: Mutable RISC-128 processor ▼

Multiprocessor: 1 CPU ▼

Opcode Type: Metamorphic - Level 3 ▼

Dynamic Opcode: 80% Dynamic ▼

VM Complexity:  Supreme

VM size (aprox):  1470 Kb

Execution Speed:  10 Mhz

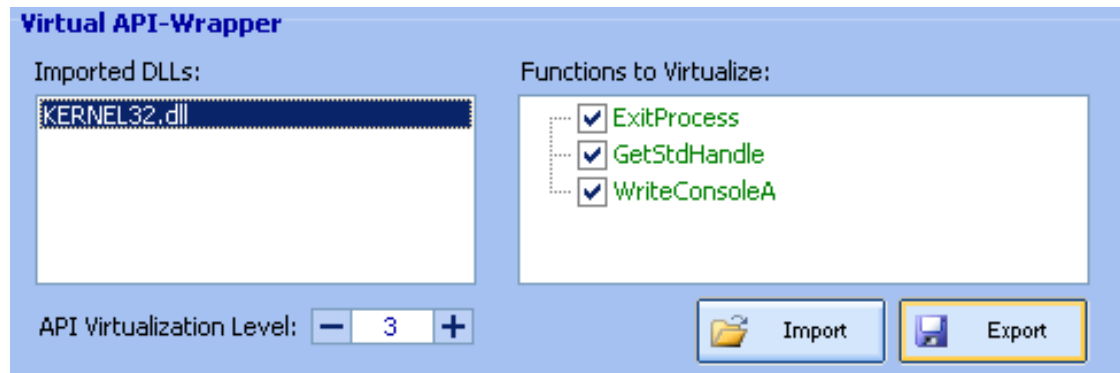
## Themida VM Options

- **Mutable RISC-128 Processor**

- “ This processor is based in RISC technology, where the size of each instruction is equal to 128 bits. Each generated RISC-128 processor will be totally different and unique (mutable) for each protected application to avoid a general attack over the embedded virtual machine. The RISC-128 processor offers higher complexity level than CISC and RISC-64 processors, but the execution performance is lower.”

(1) <http://bbs.pediy.com/showthread.php?t=50590>

## Themida API Virtualization control



- Apply metamorphic junk techniques to the bytes of the API. Note that such metamorphism is only one level deep (i.e. API called inside such API will be made as real calls).
- Copy the resulting metamorphic junks into another allocated space
  - Not virtualized, just metamorphed
- And patch the call table to points to the allocated space
- Since the API code is morphed to another address space, it would render normal API hooking/breaking useless.

## ExeCryptor – working out dispatcher

- We hypothesise that:
  - junk read and write will access the same I/O area from different IP instructions
  - Real bytecode read will be at the same IP instruction but across different area to read the bytecode
- We can prove this by plotting a graph of IP vs. I/O address

# ExeCryptor

- The virtual machine is about 15k in size
- Junk jumps similar to Themida
  - utilize more conditional obfuscation

