VMHunt: A Verifiable Approach to Partially-Virtualized Binary Code Simplification

ACM CCS 2018

Dongpeng Xu, Jiang Ming, Yu Fu, Dinghao Wu

김영철

2019. 3. 14.

Introduction

- Virtualization
  - 코드를 복잡하게 만들어..
  - 분석하기 어렵게 만들어..
  - 하지만 전체 프로그램에 대해 적용하면 원래의 코드에 비해 성능이 안 좋아져..
  - 부분적으로 적용하는 것이 최고다..

Introduction

- reverse engineer the bytecode interpreter
    - a central loop : code fetch
- strip off the virtualization obfuscation layer from the tedious execution instructions

Assume the scope of virtualization-obfuscated code is already known.

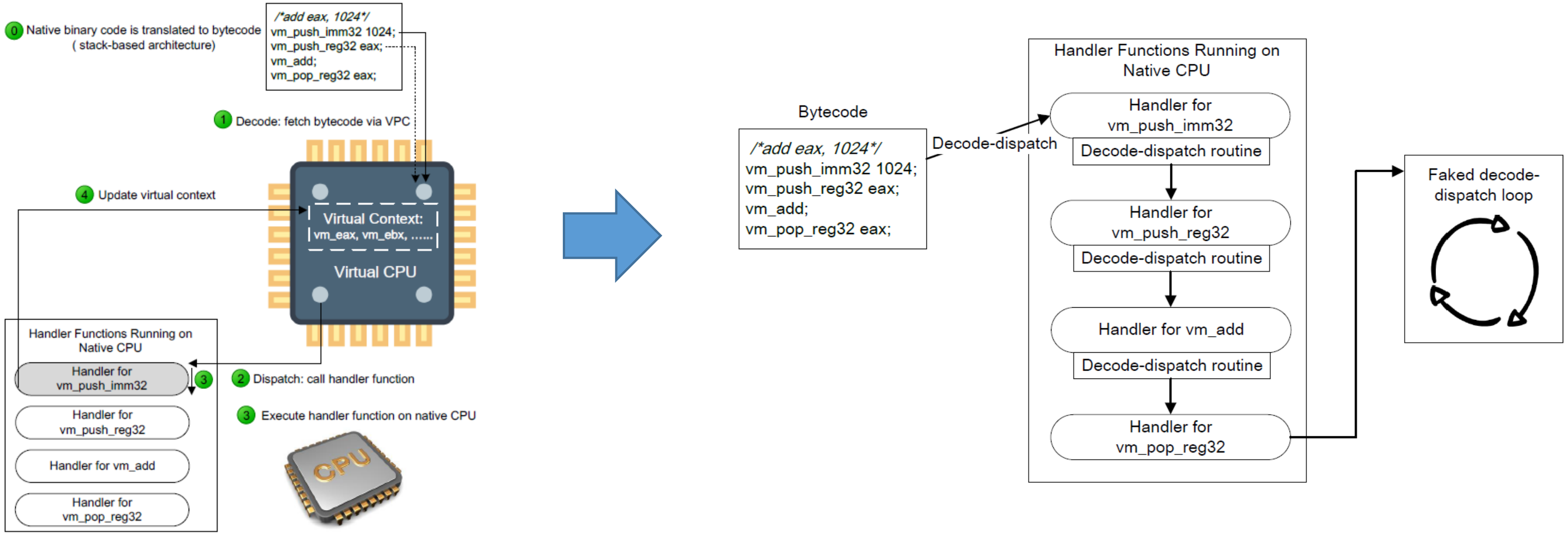
➔ Automatic detection of the virtualized code is an indispensable step

Introduction

- VMHunt
  - can detect the virtualized code section from a program execution trace
  - design a new optimization method to simplify the execution trace based on boundary information
  - be capable of performing correctness testing to the deobfuscation results

Background and Motivation

- Threaded Interpretation
  - 연구자들한테 decode-dispatch 구조가 매우 잘 알려져있음.
  - central decode-dispatch loop routine 제거



" VPC" in step 1 is short for Virtual Program Counter

Background and Motivation

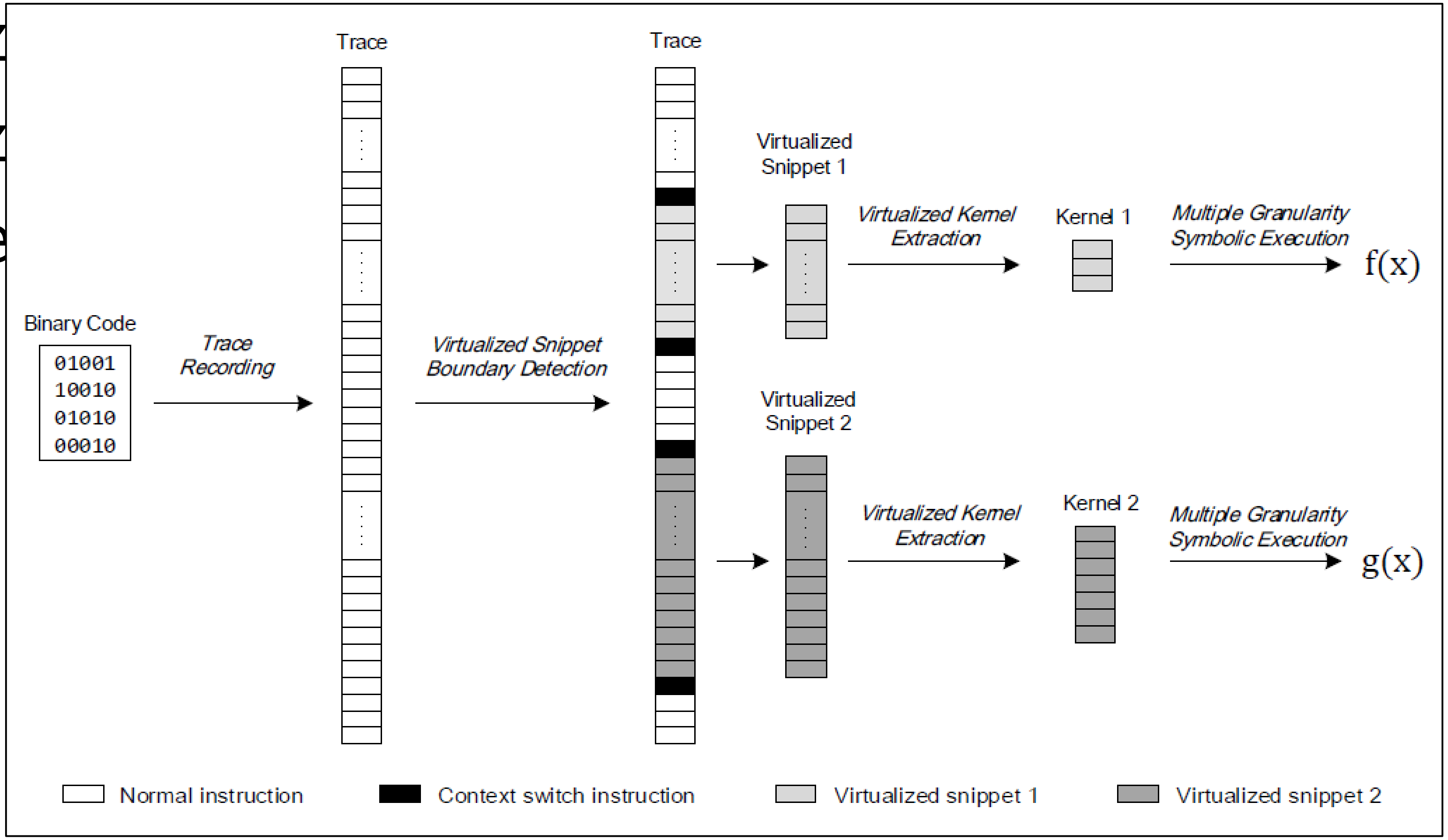
- Virtualization-Obfuscated Malware
    - 프로그램 전체를 가상화 하는 것은 사이버 범죄자들에게 좋은 선택은 아님.
    - 높은 CPU 사용률에 대한 감시를 하면 걸리기 때문.
    - 핵심 부분만 가상화 하는 것이 필수임.

Overview – VMHunt's workflow

- Virtualized Snippet Boundary Detection.
- Virtualized Kernel Extraction.
- Multiple Granularity Symbolic Execution

Overview – VMHunt's workflow

- Virtualiz
- Virtualiz
- Multiple



Virtualized Trace Boundary Detection

- Trace Logging
  - Pin tool을 이용한 dynamic tracing
    시스템 콜을 제외한 모든 명령어를 기록

  - trace information
    (1) The memory address of every instruction
    (2) The instruction name (opcode)
    (3) The source and destination operands
    (4) Runtime information (register, memory accessing address)

Virtualized Trace Boundary Detection

- Context Switch Instructions
  - 계속 언급되듯이 모든 영역을 가상화 하는 것은 비효율적.
  - 핵심이 되는 코드 영역만 부분적으로 가상화 함.
    - native 환경과 virtual 환경 간의 context switching이 존재할 것이고 탐지할 것.

```
1  ...             // native program execution
2  push  edi       // context saving
3  push  esi
4  push  ebx
5  push  edx
6  push  ebp
7  push  ecx
8  push  eax
9  pushfd
10 jmp   0x1234
11 ...             // virtualized snippet begin
12 mov   ...
13 add   ...
14 xor   ...
15 ...             // virtualized snippet end
16 popfd
17 pop   eax       // context restoring
18 pop   ecx
19 pop   ebp
20 pop   edx
21 pop   ebx
22 pop   esi
23 pop   edi
24 jmp   0x8048123
25 ...             // continue native program
26 ...             // execution
```

```
2-9   line : context saving

11-15 line : virtualized snippet

17-23 line : context restoring
```

Virtualized Trace Boundary Detection

- Context Switch Instructions

```
push  esi,ecx,edx
push  eax
push  ebp
add   esp, 0x4
sub   esp, 0x4
mov   ebp, esp
mov   eax, 0x3ff90adc
sub   eax, 0x3ff90ad8
sub   ebp, eax
add   eax, 0x4
xor   ebp, esp
xor   esp, ebp
xor   ebp, esp
mov   [esp], edi
mov   ebp, eax
push  ebx
...
```

(a) Obfuscated context switch instructions

```
push  esi,ecx,edx
sub   esp, 0x4
mov   [esp], eax
sub   esp, 0x4
mov   [esp],ebp

add   esp, 0x4
sub   esp, 0x4

mov   ebp, esp
mov   eax, 0x3ff90adc
sub   eax, 0x3ff90ad8
sub   ebp, eax

add   eax, 0x4

xor   ebp, esp
xor   esp, ebp
xor   ebp, esp

mov   [esp], edi
mov   ebp, eax

sub   esp, 0x4
mov   [esp], ebx
...
```

(b) Normalization

push esi,ecx,edx
sub esp, 0x4
mov [esp], eax
sub esp, 0x4
mov [esp],ebp

~~add esp, 0x4~~
~~sub esp, 0x4~~

~~mov ebp, esp~~
~~mov eax, 0x3ff90adc~~
~~sub eax, 0x3ff90ad8~~
~~sub ebp, eax~~
~~sub ebp, 0x4~~

add eax, 0x4

~~mov t, esp~~
~~mov esp, ebp~~
~~mov ebp, t~~

~~mov ebp, esp~~
sub esp, 0x4

mov [esp], edi
mov ebp, eax

sub esp, 0x4
mov [esp], ebx
...

(c) Simplification

```
push  esi,ecx,edx
sub   esp, 0x4
mov   [esp], eax
sub   esp, 0x4
mov   [esp],ebp
sub   esp, 0x4
mov   [esp], edi
sub   esp, 0x4
mov   [esp], ebx

add   eax, 0x4
mov   ebp, eax
...
```

Context Switch

(d) Clustering

Virtualized Trace Boundary Detection

- Context Switch Instructions
  - Normalization

```
push  esi,ecx,edx
push  eax
push  ebp
add   esp, 0x4
sub   esp, 0x4
mov   ebp, esp
mov   eax, 0x3ff90adc
sub   eax, 0x3ff90ad8
sub   ebp, eax
add   eax, 0x4
xor   ebp, esp
xor   esp, ebp
xor   ebp, esp
mov   [esp], edi
mov   ebp, eax
push  ebx
...
```

```
push  esi,ecx,edx
sub   esp, 0x4
mov   [esp], eax
sub   esp, 0x4
mov   [esp],ebp

add   esp, 0x4
sub   esp, 0x4

mov   ebp, esp
mov   eax, 0x3ff90adc
sub   eax, 0x3ff90ad8
sub   ebp, eax

add   eax, 0x4

xor   ebp, esp
xor   esp, ebp
xor   ebp, esp

mov   [esp], edi
mov   ebp, eax

sub   esp, 0x4
mov   [esp], ebx
...
```

Virtualized Trace Boundary Detection

- Context Switch Instructions
  - Simplification
    - a peephole optimizer & a data flow analyzer
      - peephole optimizer : use patter matching
      - data flow analyzer : records def-use information, perform constant propagation

Virtualized Trace Boundary Detection

- Context Switch Instructions
  - Simplification

```
 1  push  esi,ecx,edx
 2  sub   esp, 0x4
 3  mov   [esp], eax
 4  sub   esp, 0x4
 5  mov   [esp],ebp
 6
 7  add   esp, 0x4
 8  sub   esp, 0x4
 9
10  mov   ebp, esp
11  mov   eax, 0x3ff90adc
12  sub   eax, 0x3ff90ad8
13  sub   ebp, eax
14
15  add   eax, 0x4
16
17  xor   ebp, esp
18  xor   esp, ebp
19  xor   ebp, esp
20
21  mov   [esp], edi
22  mov   ebp, eax
23
24  sub   esp, 0x4
25  mov   [esp], ebx
26  ...
```

	1	push esi,ecx,edx
	2	sub esp, 0x4
	3	mov [esp], eax
	4	sub esp, 0x4
	5	mov [esp],ebp
	6	
redundant	7	~~add esp, 0x4~~
	8	~~sub esp, 0x4~~
	9	
const propagation	10	~~mov ebp, esp~~
	11	~~mov eax, 0x3ff90adc~~
	12	~~sub eax, 0x3ff90ad8~~
	13	~~sub ebp, eax~~
	14	~~sub ebp, 0x4~~
	15	
	16	add eax, 0x4
	17	
swap	18	~~mov t, esp~~
	19	~~mov esp, ebp~~
	20	~~mov ebp, t~~
	21	
not used	22	~~mov ebp, esp~~
	23	sub esp, 0x4
	24	
	25	mov [esp], edi
	26	mov ebp, eax
	27	
	28	sub esp, 0x4
	29	mov [esp], ebx
	30	...

`line 14    : 갑자기 생김`

`line 22-23 : 갑자기 생김`

peephole optimizer

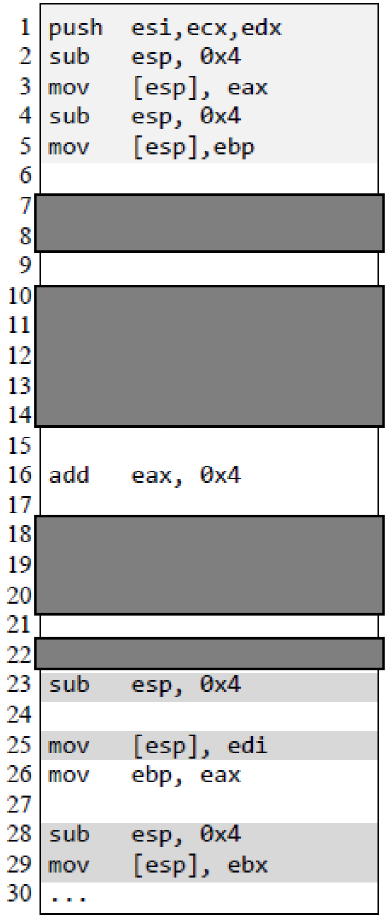
data flow analyzer

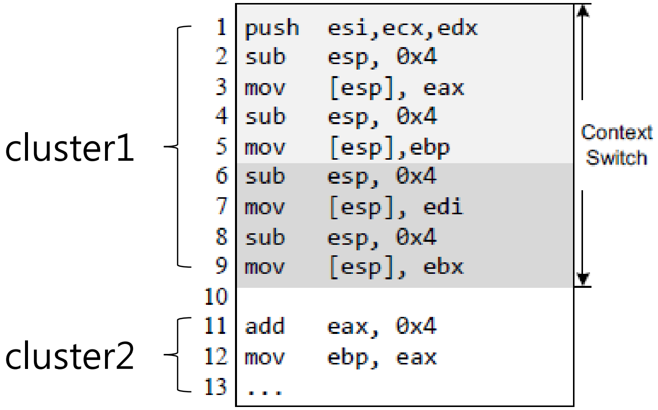
Virtualized Trace Boundary Detection

- Context Switch Instructions
  - Clustering
    - simplified trace 에서 instruction dependency graph 생성
    - dependency가 있는 명령어끼리 묶어서 그룹으로 나눔

Virtualized Trace Boundary Detection

- Context Switch Instructions





Virtualized Trace Boundary Detection

- Pairing Context Switch Instructions
  - 앞에서 context saving instructions 를 탐지함.
  - 다음은 context restoration instructions 와 pairing하여 가상화된 부분을 식별할 수 있게 됨.

  - Two heuristics
    - stack depth & execution transfer instruction

      (1) 가상화 전과 가상화 복구 후의 stack depth는 같아야 함.

      (2) context switch 는 jmp, call, ret 등과 같이 제어 흐름을 전이 시키는 명령어 근처에 존재함.

Virtualized Trace Boundary Detection

- Pairing Context Switch Instructions

stack depth at point A : N

stack depth of virtualized snippet : N + 1

stack depth at point B : N

A

Execution Transfer Instruction

Execution Transfer Instruction

B

```
...                       // native program execution
push   edi                // context saving
push   esi
push   ebx
push   edx
push   ebp
push   ecx
push   eax
pushfd
jmp    0x1234
...                       // virtualized snippet begin
mov    ...
add    ...
xor    ...
...                       // virtualized snippet end
popfd
pop    eax                // context restoring
pop    ecx
pop    ebp
pop    edx
pop    ebx
pop    esi
pop    edi
jmp    0x8048123
...                       // continue native program
...                       // execution
```

Extraction of Virtualized Kernel

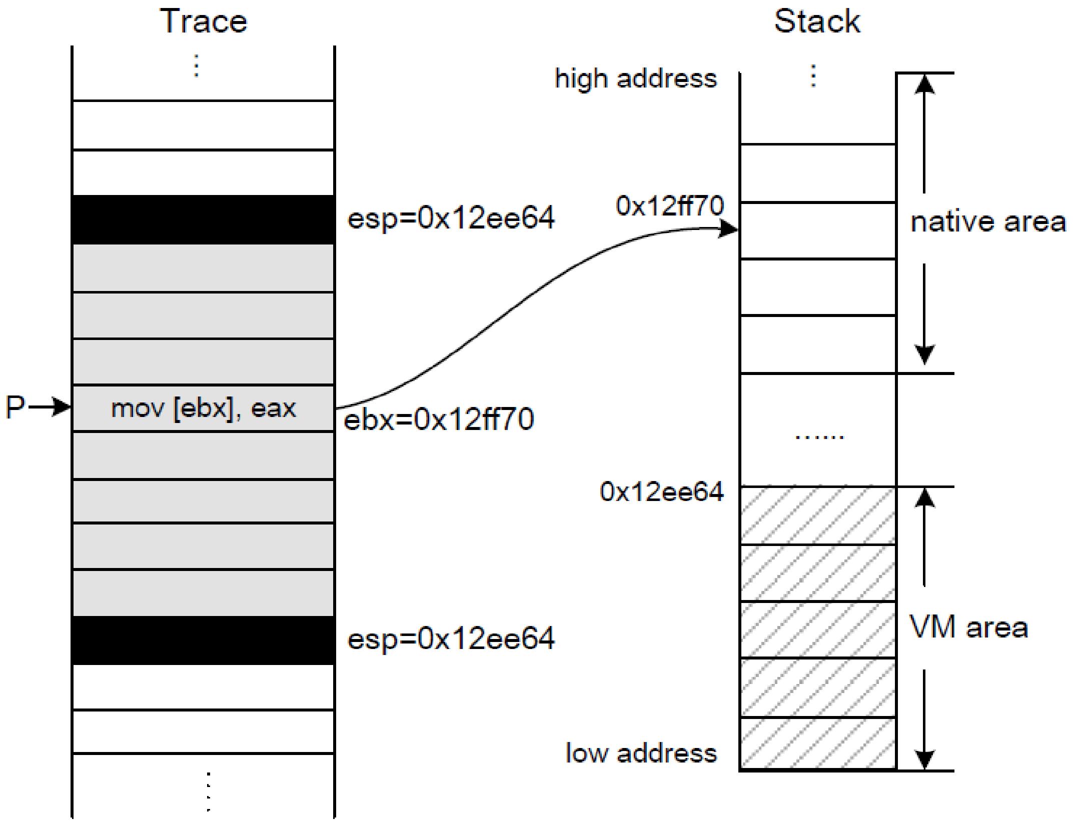
- Two types of virtualized snippet behaviors
  - local behavior : 가상화 환경에만 영향을 주는 행동
  - global behavior : native 환경에 접근하거나 영향을 주는 행동

    ➔ real function 의 정보를 얻을 수 있음.

    ➔ 이와 관련된 명령어들도 중요한 정보

Extraction of Virtualized Kernel

- Two types of virtualized snippet behaviors



At point P,
if ebx = 0x12ff70,
then [ebx] indicates in native area.

thus, {mov [ebx], eax} instruction is global behavior.

Extraction of Virtualized Kernel

- Virtualized kernel : global behavior 와 관련된 instructions
  - (1) The instruction to write data to the native area
    - ➔ directly passing
  - (2) The instruction to save data to the stack memory that will be swapped to registers by context switch.
    - ➔ indirectly passing

Extraction of Virtualized Kernel

- Backward slicing to extract the kernel
  - apply BinSim's backward slicing algorithm.
  - be able to handle many complicated issues (e.g. implicit branch logic).
  - can significantly remove unnecessary instructions.

BinSim

- Backward slicing algorithm

  step1) do the system call sequences alignment

  ➔ to get a list of matched system call pairs

  step2) conduct backward slicing on each logged trace

  ➔ to identify instructions that affect the argument

  step3) compute the weakest precondition (WP)

  ➔ WP is a symbol formula

  ➔ identify the possible cryptographic functions

  step4) use a constraint solver to verify whether two WPs are equivalent

  step5) perform an approximate matching on identified cryptographic functions, calculate the final similarity score

BinSim

(a)

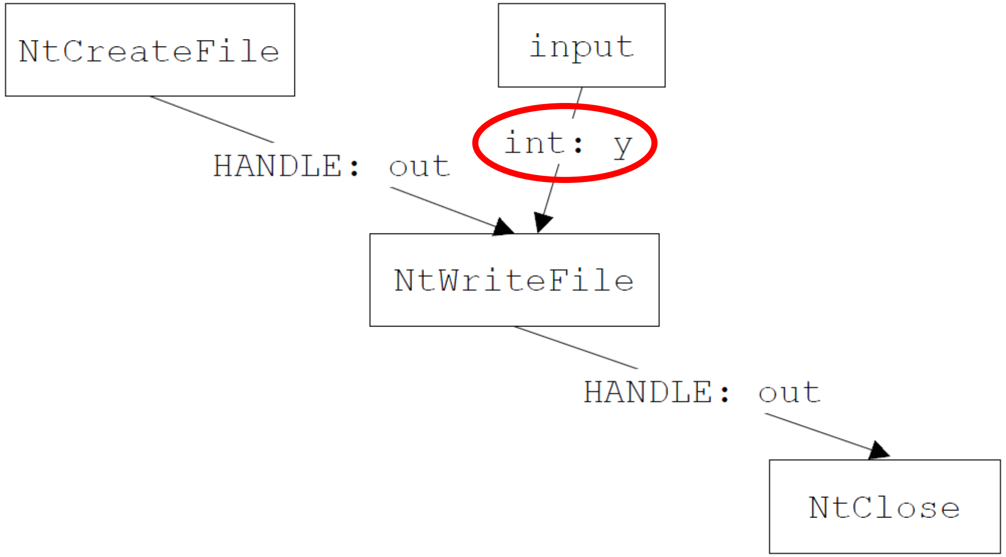
```
int x, y; // x is an input
HANDLE out = CreateFile("a.txt", ...);
y = x + x;
WriteFile(out, &y, sizeof y, ...);
CloseHandle(out);
```

(b)

```
int x, y, z; // x is an input
HANDLE out = CreateFile("a.txt", ...);
z = (x >> 31);
z = (x ^ z) - z; // z is the absolute value of x
y = 2 * z;
WriteFile(out, &y, sizeof y, ...);
CloseHandle(out);
```

(c)

```
int x, y; // x is an input
HANDLE out = CreateFile ( "a.txt", ... );
y = x << 1;
WriteFile ( out, &y, sizeof y, ... );
CloseHandle ( out );
```



<the aligned system call sequences>

BinSim

(a)

```
int x, y; // x is an input
HANDLE out = CreateFile("a.txt", ...);
y = x + x;
WriteFile(out, &y, sizeof y, ...);
CloseHandle(out);
```

$$Formula_a = x + x$$

(b)

```
int x, y, z; // x is an input
HANDLE out = CreateFile("a.txt", ...);
z = (x >> 31);
z = (x ^ z) - z; // z is the absolute value of x
y = 2 * z;
WriteFile(out, &y, sizeof y, ...);
CloseHandle(out);
```

$$Formula_b = 2 \ast ((x \wedge (x \gg 31)) - (x \gg 31))$$

(c)

```
int x, y; // x is an input
HANDLE out = CreateFile ( "a.txt", ... );
y = x << 1;
WriteFile ( out, &y, sizeof y, ... );
CloseHandle ( out );
```

$$Formula_c = x \ll 1$$

BinSim

$$Formula_a = x + x$$

(a) and (c) are truly matched

(a) and (b) *conditionally equivalent*
(when the input satisfies x≥0)

$$Formula_b = 2 * ((x \wedge (x \gg 31)) - (x \gg 31))$$

$$Formula_c = x \ll 1$$

Multiple Granularity Symbolic Execution

- extract the semantics of the virtualized code by symbolic execution.
- can verify whether our deobfuscation is a semantically equivalent translation.

- Modern virtualized code usually come with lots of bit-wise operations.
- Traditional SE fails to optimize the formulas which contain symbols in different granularity.

Multiple Granularity Symbolic Execution

- Multiple Granularity Symbolic Execution
  - the length of symbols is not fixed
  - can perfectly optimize formulas
  - can generate neat formulas

    ➔ be easily handled by theorem solvers
  - “interpret” the semantics, rather than “translate”

Multiple Granularity Symbolic Execution

- Multiple Granularity Symbolic Execution
  (1) maintain runtime status of registers
  (2) interpret the effect of each instruction, e.g., shl/shr
  (3) remove redundant symbols if they become concrete values after the interpretation of the instruction

Multiple Granularity Symbolic Execution

- Def 6.1. two types of value: concrete & symbolic

  C_n^m : id is *n*, *m*-bit concrete value

- Def 6.2. $[v_1, .., v_n]$ is to represent the concatenation of n values.

  $[S_1^8, C_1^8, C_2^8]$ means a concatenation of the three 8-bit values (one symbolic value & two concrete values).

- Def 6.3. The symbol | is used for binding value

  $[S_1^8, C_2^8]|_{C_2^8=0x23}$ means that C_2^8 is bound to 0x23.

Multiple Granularity Symbolic Execution

- Single
  - fixed granularity
  - redundant symbol, unoptimized symbol
  - too much time for a solver to solve them

- Multiple Granularity SE
  - handle above the limitations

Multiple Granularity Symbolic Execution

- Single vs Multiple Granularity SE
    - apply two SEs for a example
    - take processes flow
    - check for results of SEs

```
mov  eax, 0x12345678
mov  ah,  mem[0x14ff23]
shl  eax, 4
and  eax, 0x0ff00ff0
```

Multiple Granularity Symbolic Execution

- Single vs Multiple Granularity SE

```
mov  eax, 0x12345678
```

one byte single SE

$[C_4^8, C_3^8, C_2^8, C_1^8]|_{C_4^8=0x12, C_3^8=0x34, C_2^8=0x56, C_1^8=0x78}$

Multiple Granularity SE

$C_1^{32}|_{C_1^{32}=0x12345678}$

Multiple Granularity Symbolic Execution

- Single vs Multiple Granularity SE

```
mov  eax, 0x12345678
mov  ah,  mem[0x14ff23]
```

one byte single SE

$[C_4^8, C_3^8, C_2^8, C_1^8]|_{C_4^8=0x12, C_3^8=0x34, C_2^8=0x56, C_1^8=0x78}$

$[C_4^8, C_3^8, S_1^8, C_1^8]|_{C_4^8=0x12, C_3^8=0x34, C_1^8=0x78}$

Multiple Granularity SE

$C_1^{32}|_{C_1^{32}=0x12345678}$

$[C_2^{16}, S_1^8, C_3^8]|_{C_2^{16}=0x1234, C_3^8=0x78}$

Multiple Granularity Symbolic Execution

- Single vs Multiple Granularity SE

```
mov  eax, 0x12345678
mov  ah,  mem[0x14ff23]
shl  eax, 4
```

one byte single SE

$[C_4^8, C_3^8, C_2^8, C_1^8]|_{C_4^8=0x12, C_3^8=0x34, C_2^8=0x56, C_1^8=0x78}$

$[C_4^8, C_3^8, S_1^8, C_1^8]|_{C_4^8=0x12, C_3^8=0x34, C_1^8=0x78}$

$S_3^{32} = shl\left(S_2^{32}, C_5^8\right)|_{S_2^{32}=[C_4^8, C_3^8, S_1^8, C_1^8], C_5^8=0x4}$

Multiple Granularity SE

$C_1^{32}|_{C_1^{32}=0x12345678}$

$[C_2^{16}, S_1^8, C_3^8]|_{C_2^{16}=0x1234, C_3^8=0x78}$

$[C_4^{12}, S_1^8, C_5^{12}]|_{C_4^{12}=0x234, C_5^{12}=0x780}$

Multiple Granularity Symbolic Execution

- Single vs Multiple Granularity SE

```
mov  eax, 0x12345678
mov  ah,  mem[0x14ff23]
shl  eax, 4
and  eax, 0x0ff00ff0
```

one byte single SE

$[C_4^8, C_3^8, C_2^8, C_1^8]|_{C_4^8=0x12, C_3^8=0x34, C_2^8=0x56, C_1^8=0x78}$

$[C_4^8, C_3^8, S_1^8, C_1^8]|_{C_4^8=0x12, C_3^8=0x34, C_1^8=0x78}$

$S_3^{32} = shl\left(S_2^{32}, C_5^8\right)|_{S_2^{32}=[C_4^8, C_3^8, S_1^8, C_1^8], C_5^8=0x4}$

$S_4^{32} = and(S_3^{32}, C_6^{32})|_{C_6^{32}=0x0ff00ff0}$

Multiple Granularity SE

$C_1^{32}|_{C_1^{32}=0x12345678}$

$[C_2^{16}, S_1^8, C_3^8]|_{C_2^{16}=0x1234, C_3^8=0x78}$

$[C_4^{12}, S_1^8, C_5^{12}]|_{C_4^{12}=0x234, C_5^{12}=0x780}$

$[C_4^{12} \wedge 0x0ff, S_1^8 \wedge 0x00, C_5^{12} \wedge 0xff0]|_{C_4^{12}=0x234, C_5^{12}=0x780}$

$[C_6^{12}, C_7^8, C_8^{12}]|_{C_6^{12}=0x034, C_7^8=0x00, C_5^{12}=0x780}$

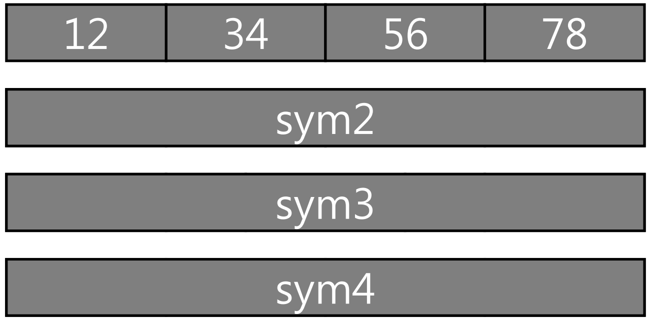
$C_9^{32}|_{C_9^{32}=0x03400780}$

Multiple Granularity Symbolic Execution

- Single vs Multiple Granularity SE

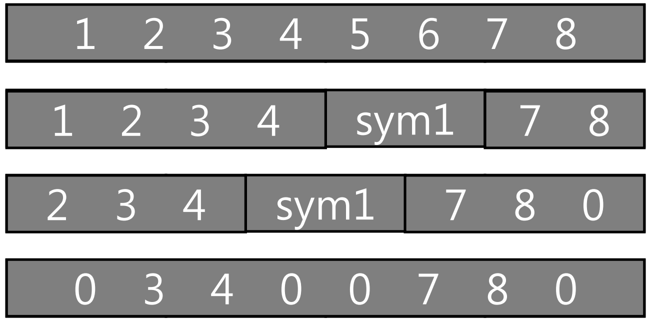
```
mov   eax, 0x12345678
mov   ah,  mem[0x14ff23]
shl   eax, 4
and   eax, 0x0ff00ff0
```

one byte single SE



6 constant values
4 symbolic values

Multiple Granularity SE



9 constant values
1 symbolic value

Multiple Granularity Symbolic Execution

- In summary,
  (1) Fine-grained Analysis
    - accurately interpret the semantics of bitwise operations
    - available to optimize for eliminating redundant symbolic values

  (2) Flexibility
    - free to split/merge values without granularity restriction

Implementation

- VMHunt
  - Pin based trace logger
  - Multiple-grained symbolic execution engines
  - Parser for lifting a trace to the IR (encode symbolic & concrete values)
  - Boundary detector
  - Slicer
  - Optimizer
  - Utilities for CFG generation
  - 17,192 lines of C++ code, 341 lines of Perl code

Evaluation

- grep-2.21
- bzip2-1.0.6
- md5sum-8.24
- AES in OpenSSL-1.1.0-pre3
- thttpd-2.26
- sqlite-2.26

- Code Virtualizer
- Themida
- VMProtect
- EXEcryptor

Evaluation

- VMProtect and EXECryptor are similar.
- Themida and Code Virtualizer is the same.

- The virtualized snippet identified by VMHunt is about 10% of the whole trace size.
- The kernel is about 10^{-4} of the whole trace size.
  - ➔ VMHunt can significantly reduce the size of trace.

Evaluation

- test data가 뭔지 잘 모르겠음..
- 저 프로그램들이 가상화 되어있다는 건지..? 맞는 것 같음.

Programs	T	S1	S2	S1+S2	K1	K2	K1+K2	(S1+S2)/T(%)	(K1+K2)/T(10^{-4})
grep	1,072,446	130,329	168,857	299,186	552	1,061	1,613	24.6	15.0
bzip2	1,422,428	133,272	153,537	286,809	774	1,444	2,218	20.2	15.6
aes	2,479,948	124,793	156,019	280,812	837	1,173	2,010	11.3	8.1
md5sum	2,309,826	134,320	168,163	302,483	604	1,271	1,875	13.1	8.1
thttpd	3,680,610	117,435	155,262	272,697	677	1,389	2,066	7.4	5.6
sqlite	4,716,883	146,177	161,073	307,250	820	1,465	2,285	6.5	4.8

Evaluation

- 1-byte vs 1-bit vs Multiple Granularity SE

SE	Metrics	grep	bzip2	aes	md5sum	thttp	sqlite
byte	size	671	459	674	801	792	997
	var #	1289	2071	3215	4318	4730	6103
	time	90	105	150	152	144	183
bit	size	7992	5205	5310	9134	6840	10289
	var #	25110	41947	69827	87638	80592	13609
	time	383	486	532	517	793	-
MG	size	71	128	218	239	291	348
	var #	408	558	544	673	804	930
	time	12	16	13	14	20	23

Evaluation

- Multiple VMs Virtualization (nested virtualization)

Programs	T	S1	S2	S1+S2	K1	K2	K1+K2	(S1+S2)/T(%)	(K1+K2)/T(10^{-4})
grep	1,217,671	122,615	231,807	354,422	537	1,458	1,995	29.1	16.4
bzip2	1,594,486	120,103	206,049	326,152	713	1,540	2,253	20.8	14.1
aes	2,566,455	110,801	240,743	351,544	792	1,675	2,467	13.7	9.6
md5sum	2,310,301	138,508	249,138	387,646	649	1,549	2,198	16.8	9.5
thttpd	3,691,011	123,080	277,226	400,306	711	1,563	2,274	10.8	6.2
sqlite	4,764,819	143,995	294,373	438,368	802	1,898	2,700	9.2	5.7

Evaluation

- Malware samples – Botnet, Virus, Ransomware
  - be already known as being virtualized

Name	Type	T	S	K	S/W	K/W
chodebot	Botnet	1,967,000	150,129	930	5.9	4.7
nzm	Botnet	6,141,556	181,457	1432	3.0	2.3
phatbot	Botnet	2,224,405	152,723	1008	6.9	4.5
zswarm	Botnet	5,587,140	168,529	1395	3.0	2.5
tsgh	Botnet	5,199,837	145,372	1362	3.0	2.6
dllinject	Virus	8,634,893	174,232	1568	2.0	1.8
locker_builder	Virus	10,435,886	198,695	1293	1.9	1.2
locker_locker	Virus	4,594,868	146,960	893	3.2	1.9
temp_java	Virus	8,661,836	185,380	1504	2.1	1.7
tears	Ransom	2,658,615	143,308	1074	5.4	4.0

➔ the virtualized part is actually the procedure of key generation. (tears case)

Evaluation

- Unvirtualized Programs
  - Snippet 4 and 7 are real virtualization snippet.
  - the ratio between kernel and the snippet is a good metric.
  - the snippet length is also a good metric.

Snippet	S	K	K/S(%)
1	5,371	5,103	95.01
2	218	218	100.00
3	3,557	3,282	92.27
4	130,329	552	0.42
5	1,697	1,572	92.63
6	2,392	2,288	95.61
7	168,857	1061	0.63

Evaluation

- Performance
  - VMHunt : trace logging and offline analysis
  - the overhead of trace logging is about 5X slow down.

Programs	BD	K-Extraction	MGSE	Total
grep	7.2	4.8	5.3	17.3
bzip2	9.3	3.7	4.7	17.7
aes	10.9	4.1	6.3	21.3
md5sum	11.4	4.9	5.8	22.1
thttpd	14.7	4.7	5.1	24.5
sqlite	16.9	5.1	6.7	28.7

<execution time of offline analysis>

Discussion

- VMHunt's limitations & countermeasures
    - insufficient path coverage
        - automatically generate new inputs to explore uncovered paths.
    - in dynamic binary instrumentation environment
        - run malware in a transparent environment.
    - can mislead the detection of VM context switch by inserting redundant context switch instructions
        - can be removed in the simplification
        - can check whether the switched context is actually used in the kernel
    - can strengthen obfuscation by diversifying VM contexts & handler
        - semantics-based simplification is able to deal with it

Discussion

- if the whole program is virtualized, VMHunt can not detect.
  - because no context switch occur
  - but whole program virtualization rarely happens in practice
- can customize a VM that only uses some of the registers
  - so the context switch instructions would only save/restore those used registers.
  - but have not observed any VM using partial registers

E N D