Multi-Aspect Profiling of Kernel Rootkit Behavior

Ryan Riley, Xuxian Jiang, Dongyan Xu
Purdue University, North Carolina State University

EuroSys 2009
Nürnberg, Germany

Rootkits

- Stealthy malware
- Hide attacker
- Modifying the OS kernel in memory
- Injecting new code

- Threat model:
  - “Root” privileges
  - Full memory access

In the news…

USA TODAY
Classifieds:
cars.com
careerbuilder
Marketplace
Real esta
Home
News
Travel
Money
Sports
Life
Tech
Weather
Search
powered by YAHOO!
GO
Tech Products
Products home
Talking Tech
Edward C. Baig
Cyberspeak
E-MAIL THIS
PRINT THIS
SAVE THIS
MOST POPULAR
SUBSCRIBE
Posted 11/17/2005 5:00 PM
CyberSpeak
Andrew Kantor
Sony: The rootkit of all evil?

Rootkit techniques

Rootkit techniques

adore-ng

- Linux 2.4/2.6
- Kernel module
- Adds “custom” functions

Rootkit techniques

adore-ng

- Linux 2.4/2.6
- Kernel module
- Adds “custom” functions

hp

- Linux 2.4
- Kernel module
- Modifies kernel objects

Profiling a rootkit?

- Quickly reveal behavior
- Tool for malware investigators
- Honeypot environment
- This is hard, rootkits are highly privileged!

Profiling: Determining behavior

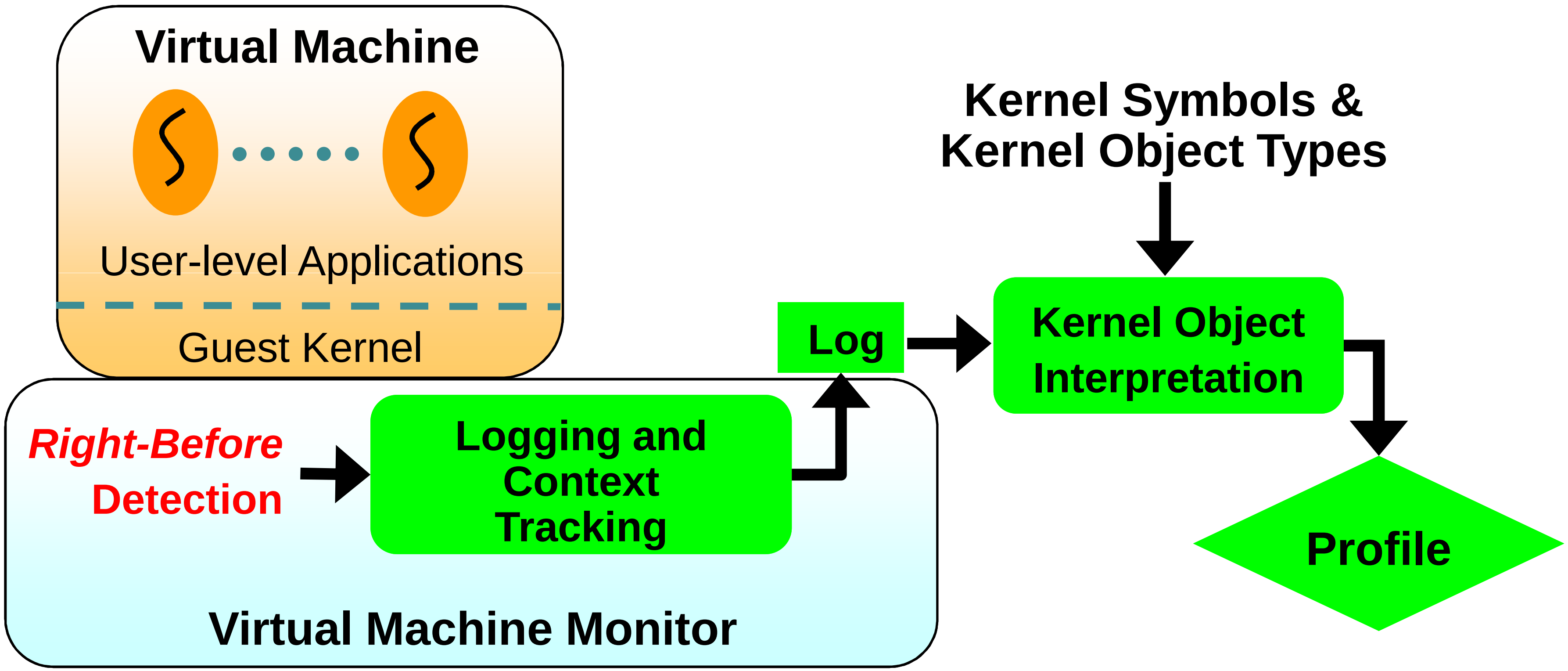
1. What code does it run?

2. What kernel objects does it modify?

3. How does it modify control flow?

4. What system calls are affected at user-level?

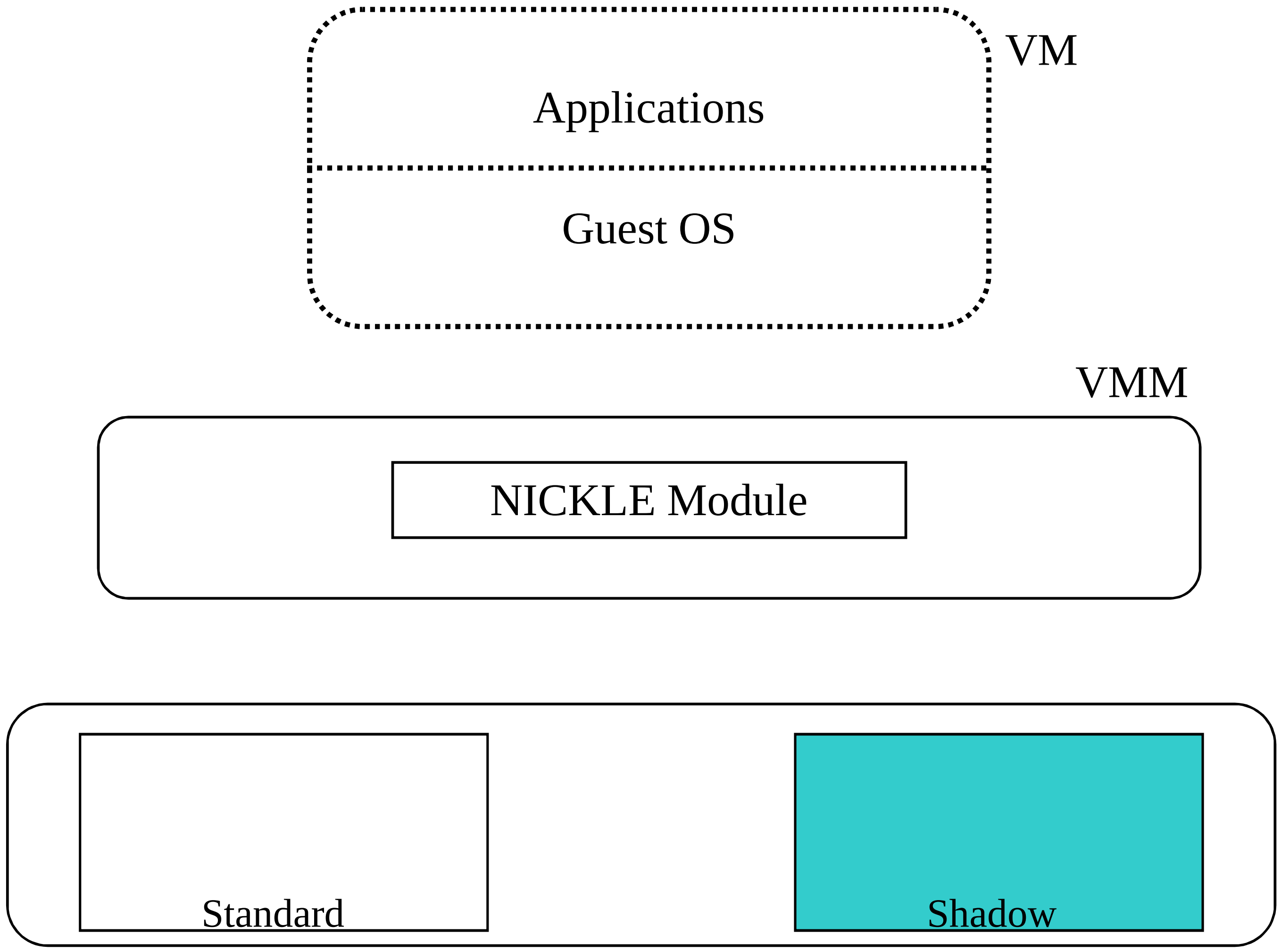
PoKeR: Architecture



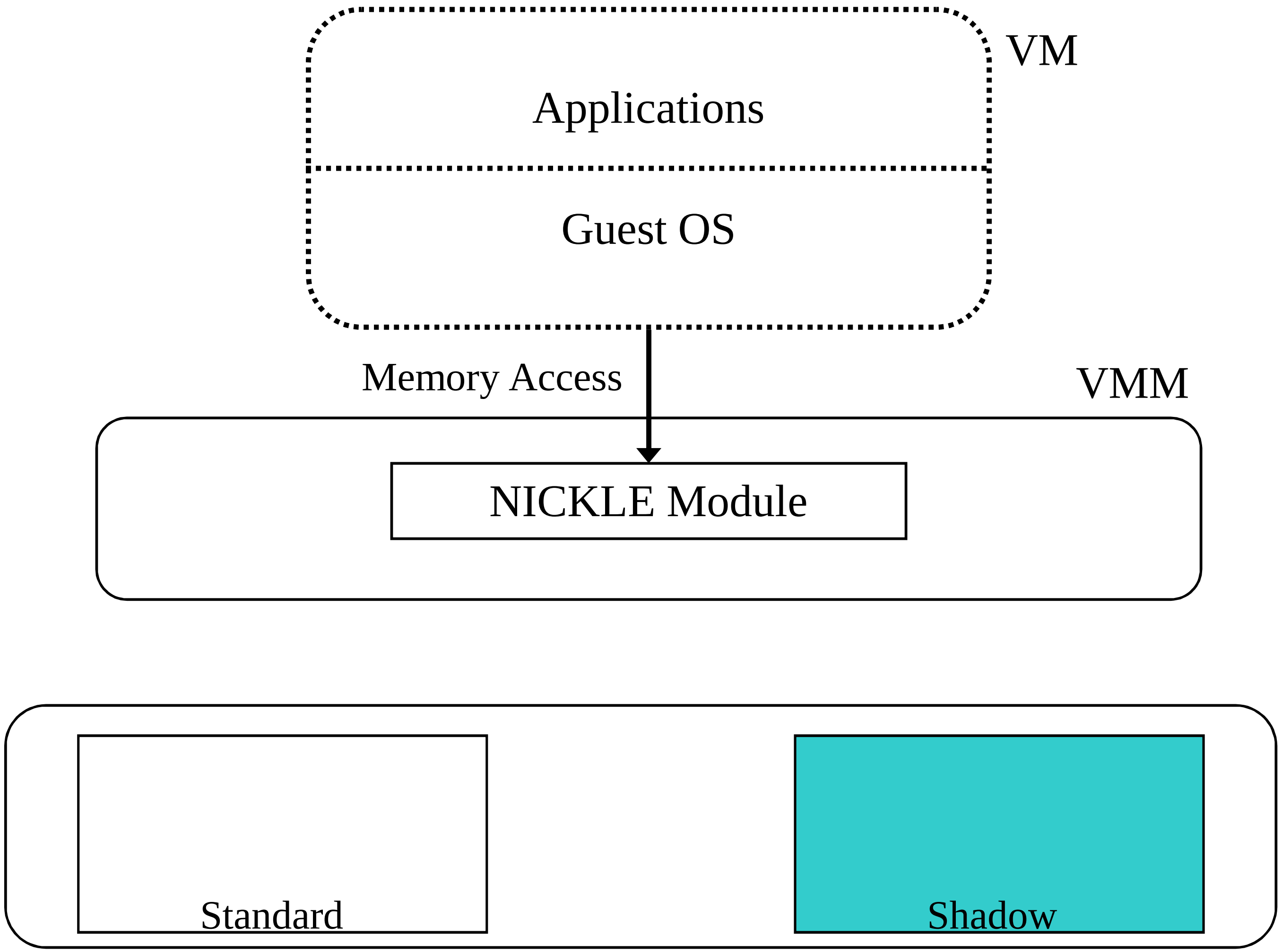
PoKeR: Architecture

***Right-Before* Detection** → **Logging and Context Tracking**

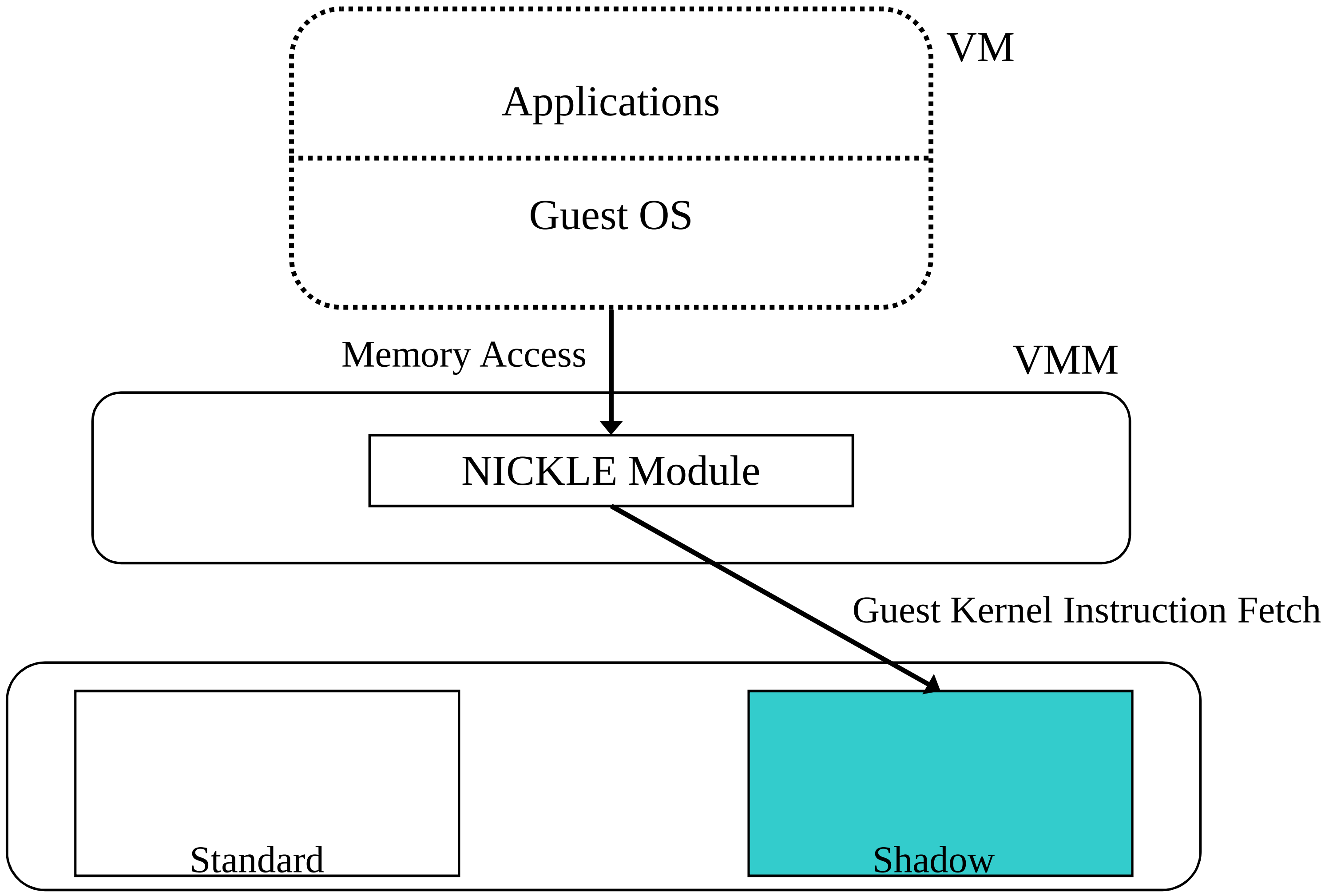
“Right before” detection?



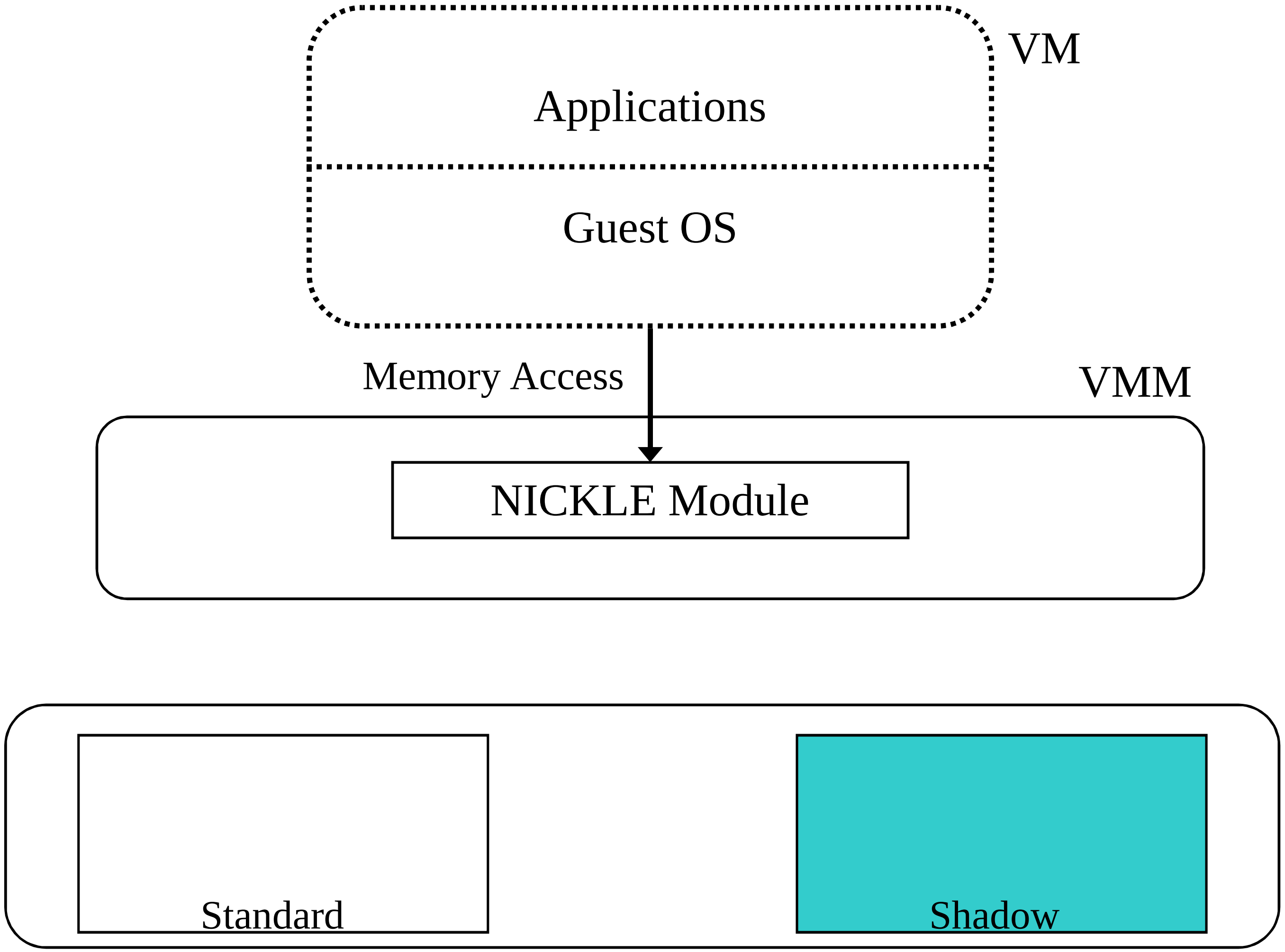
“Right before” detection?



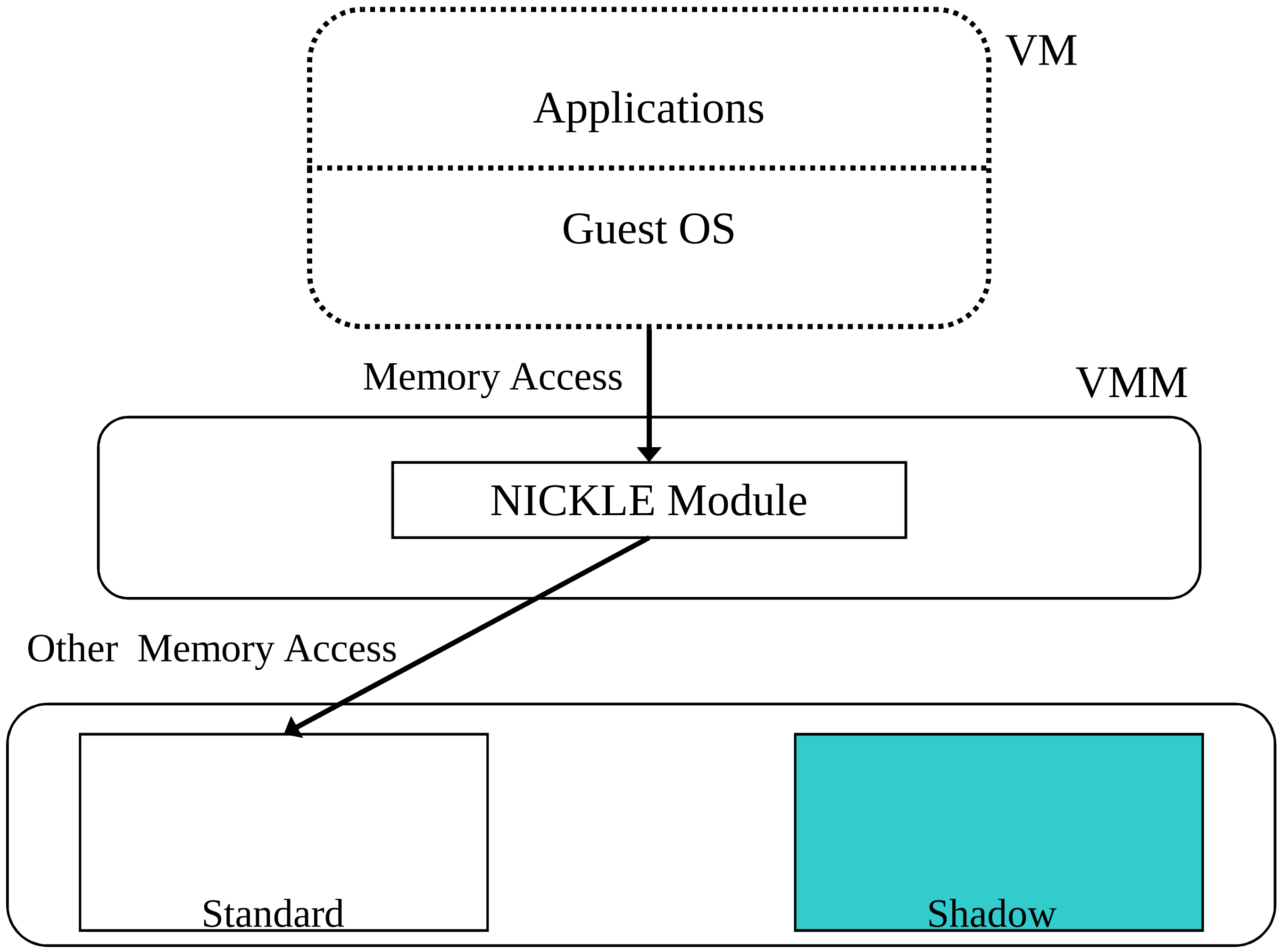
“Right before” detection?



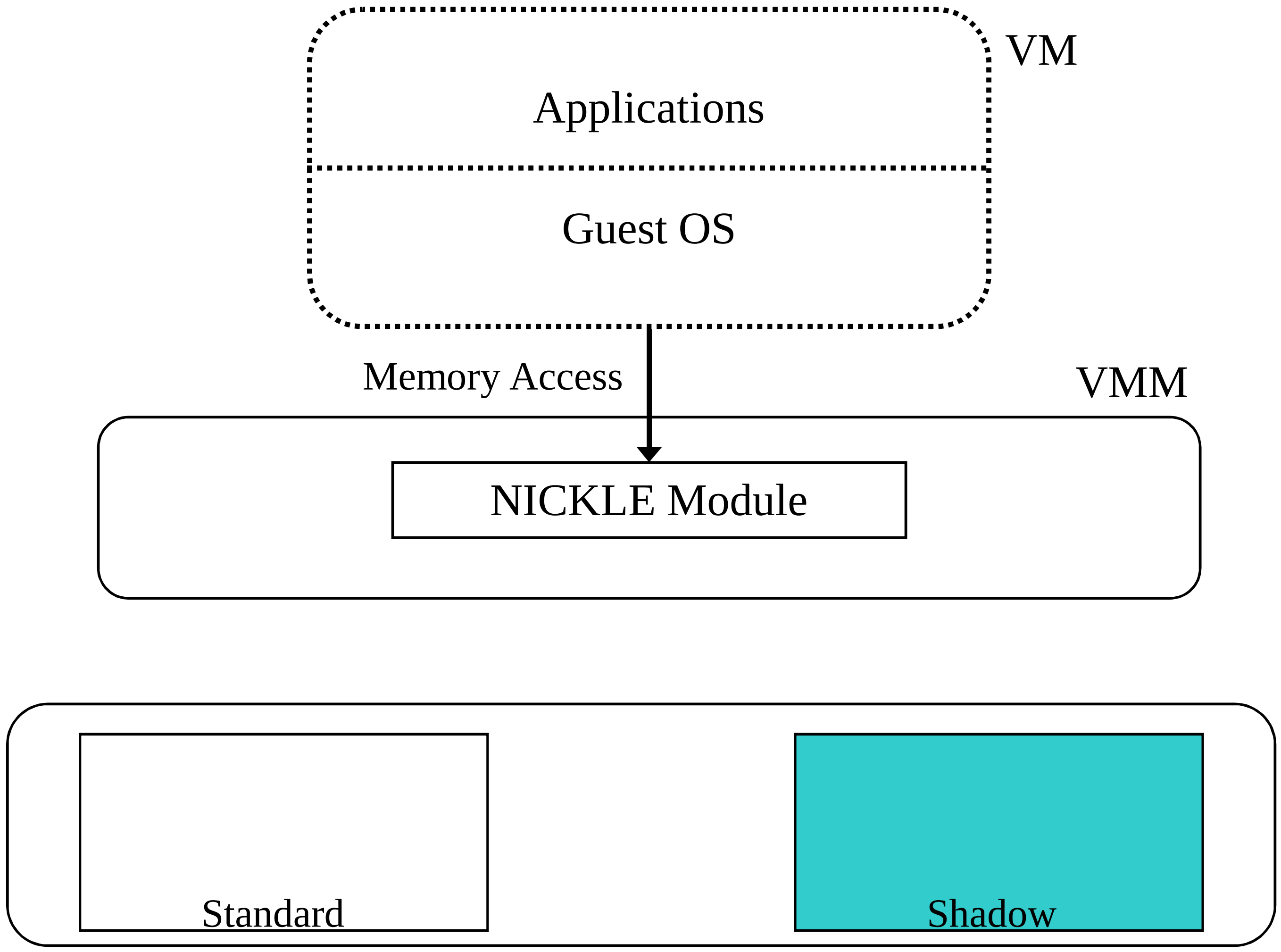
“Right before” detection?



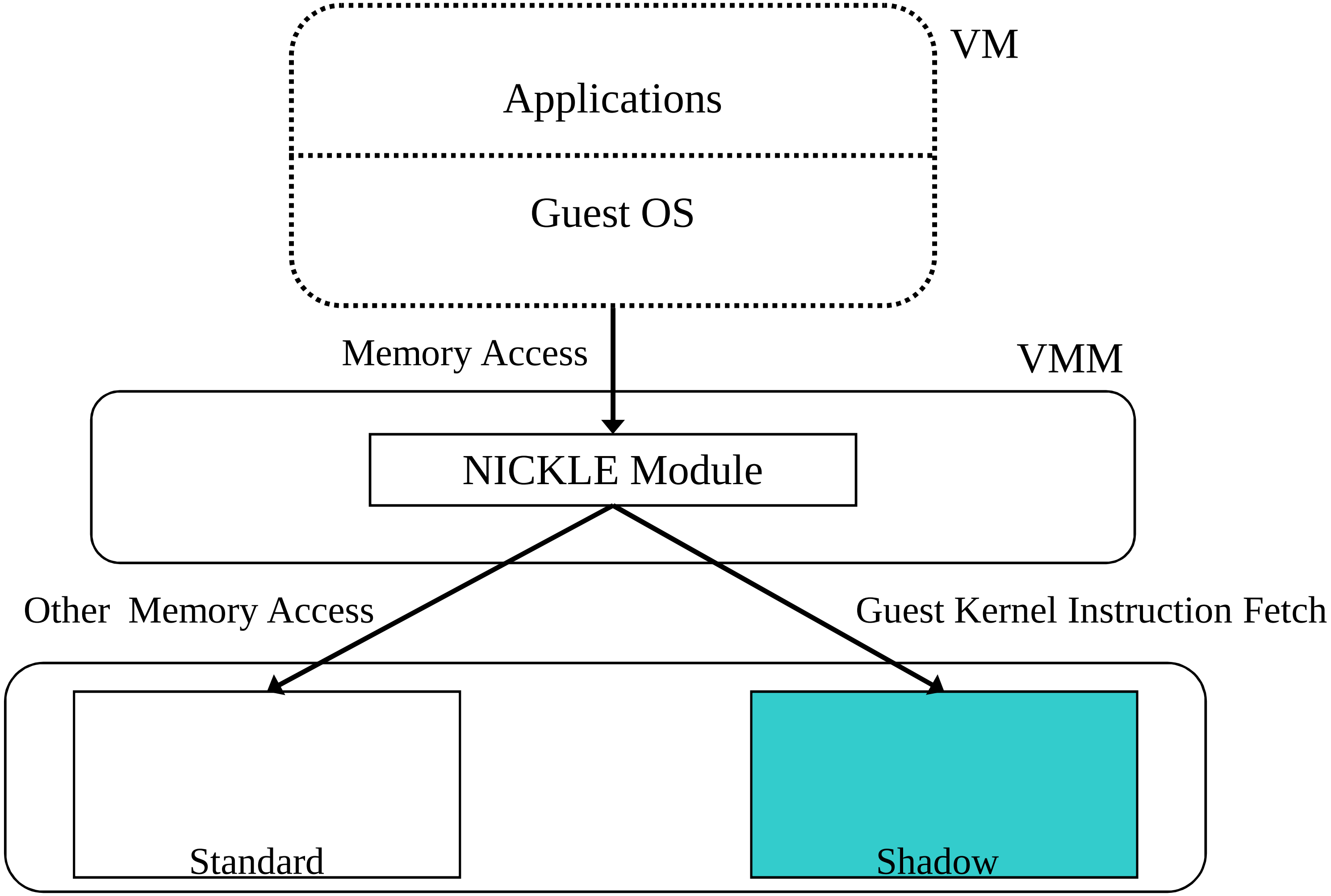
“Right before” detection?



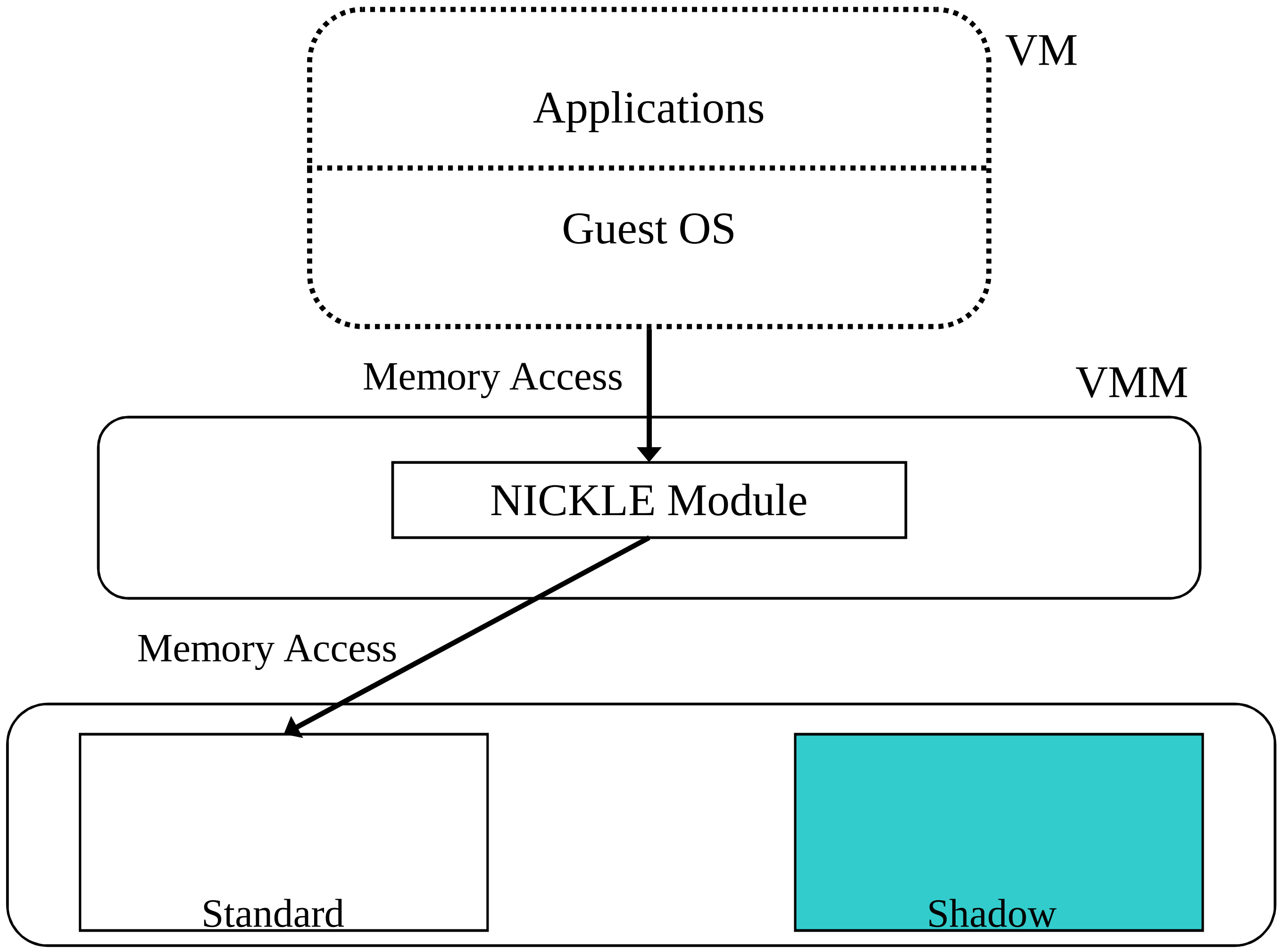
“Right before” detection?



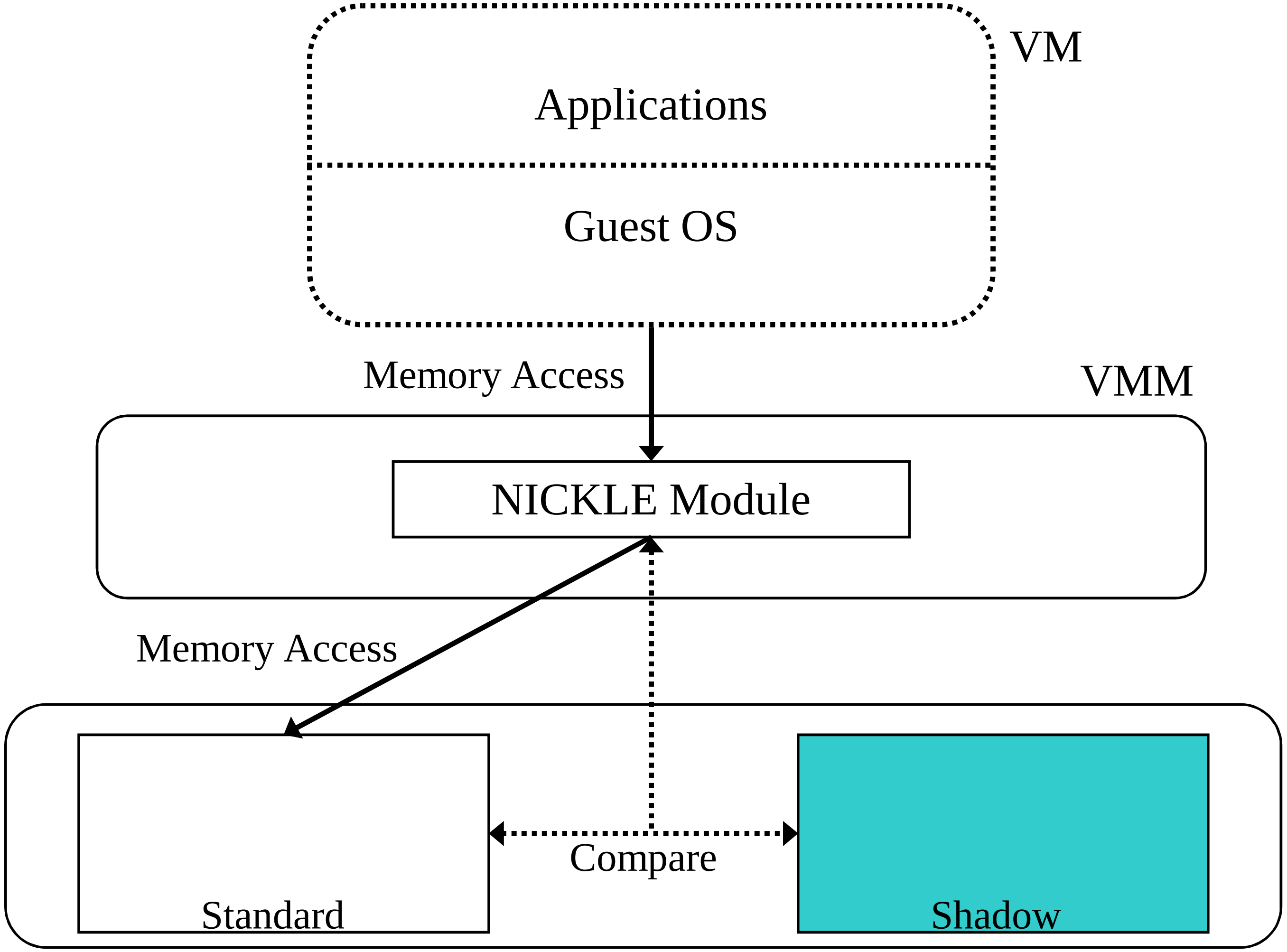
“Right before” detection?



“Right before” detection?



“Right before” detection?

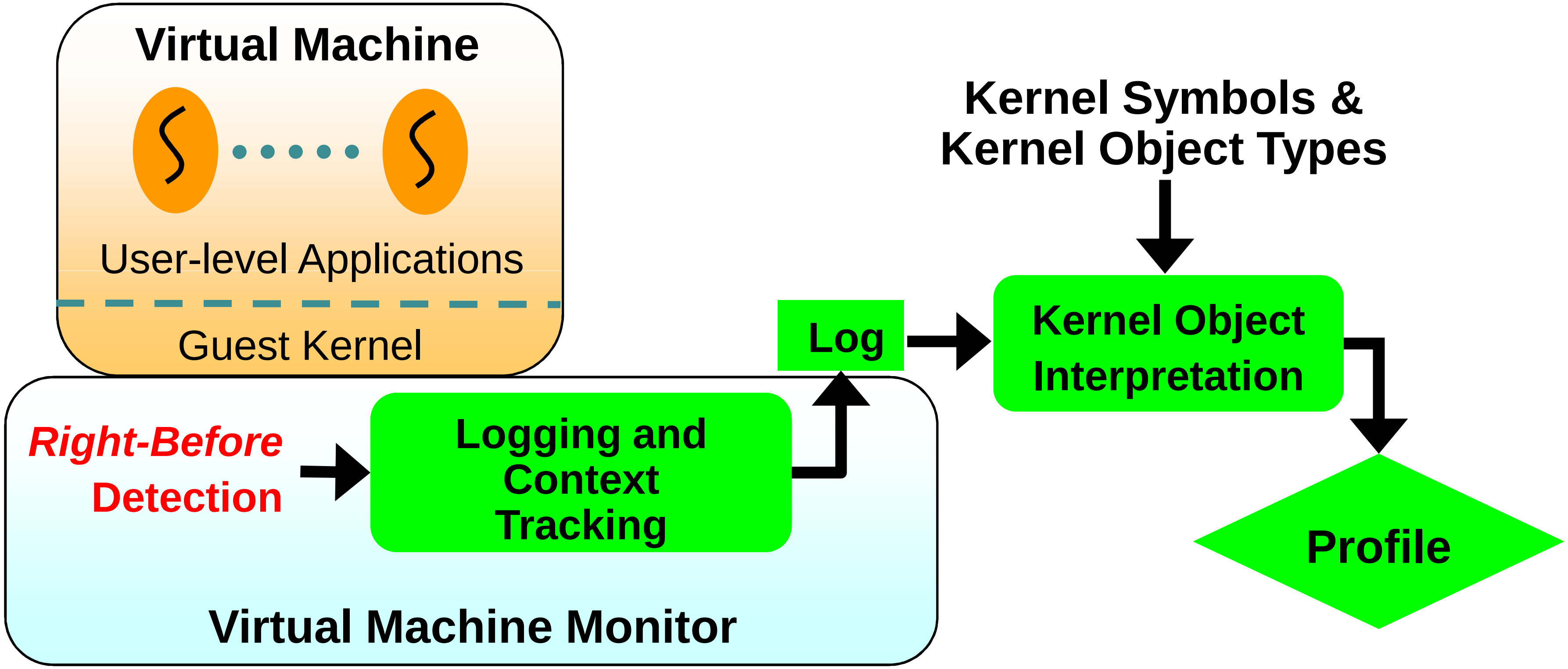


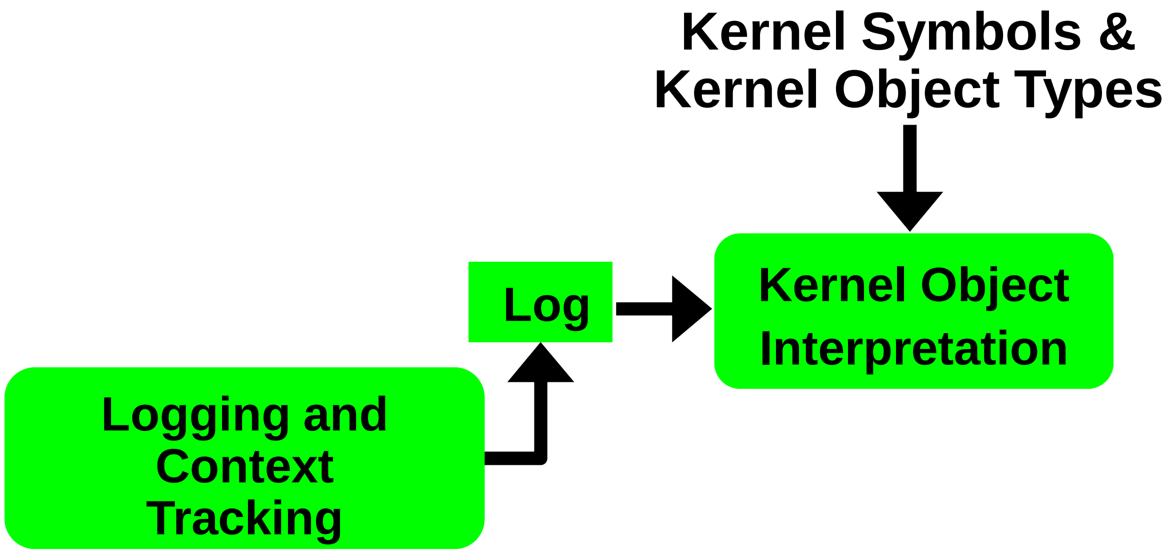
What code does it run?

- Compare standard and shadow memories
  - – Extract code as you go

```
push        %ebp
mov         %esp, %ebp
sub         $0x0C, %esp
mov         $0x00001000, %ecx
push        %edi
push        %esi
push        %ebx
movl        0x14(%ebp), %eax
mov         $0x0804EF39, %ebx
sub         $0x0804D040, %ebx
movl        0xC(%ebp), %edx
movl        %eax, 0xEC(%edx)
movl        0x8(%ebp), %esi
leal        0x400(%esi,%ebx), %esi
movl        %esi, -0x4(%ebp)
mov         $0x00, %dl
mov         %esi, %edi
mov         %dl, %al
repz stosb %al, %es:(%edi)
lcall       0xFFFFE40B
```

PoKeR: Architecture




Kernel Symbols &
Kernel Object Types
Log
Kernel Object
Interpretation
Logging and
Context
Tracking

Logging and context tracking

- Logging rootkit code…
  - Execution
  - Reads
  - Writes

What kernel objects does it modify?

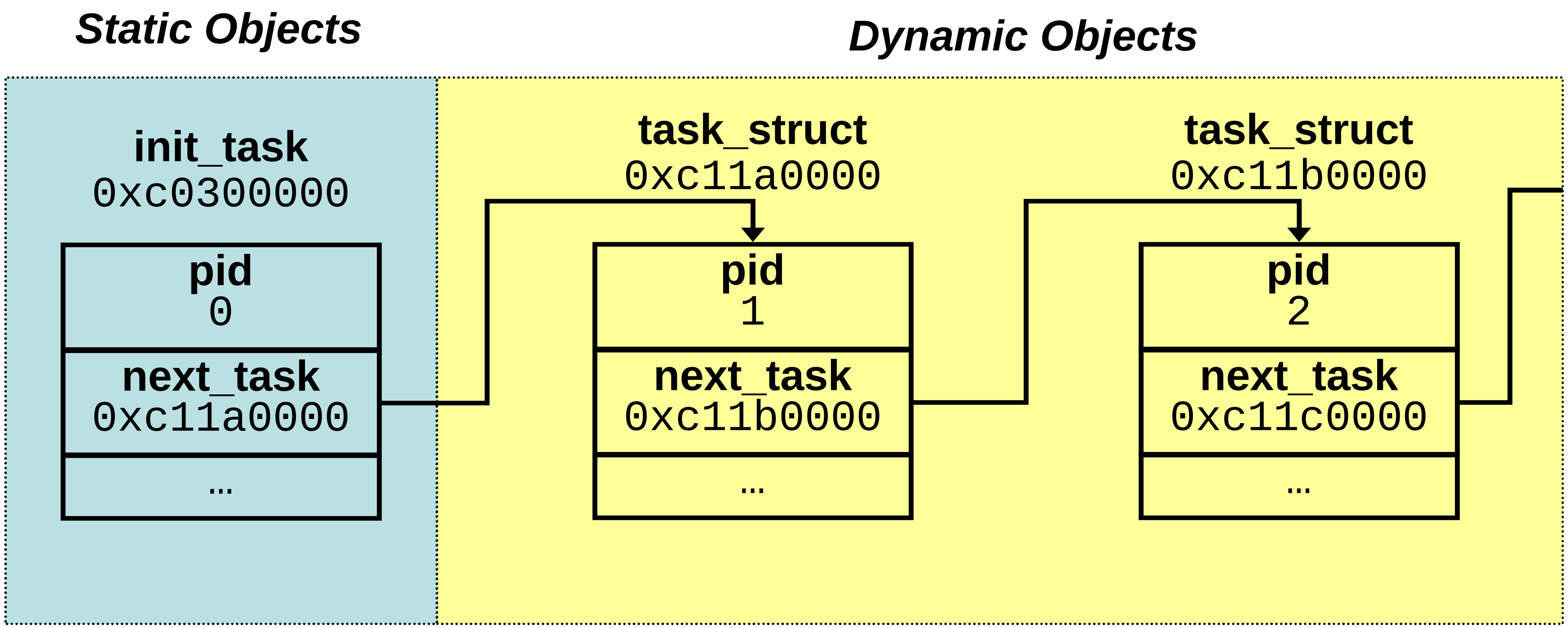
- We have memory **writes** from rootkit code
- Use static analysis to build a map
  - Kernel with debug symbols

What about dynamic allocation?

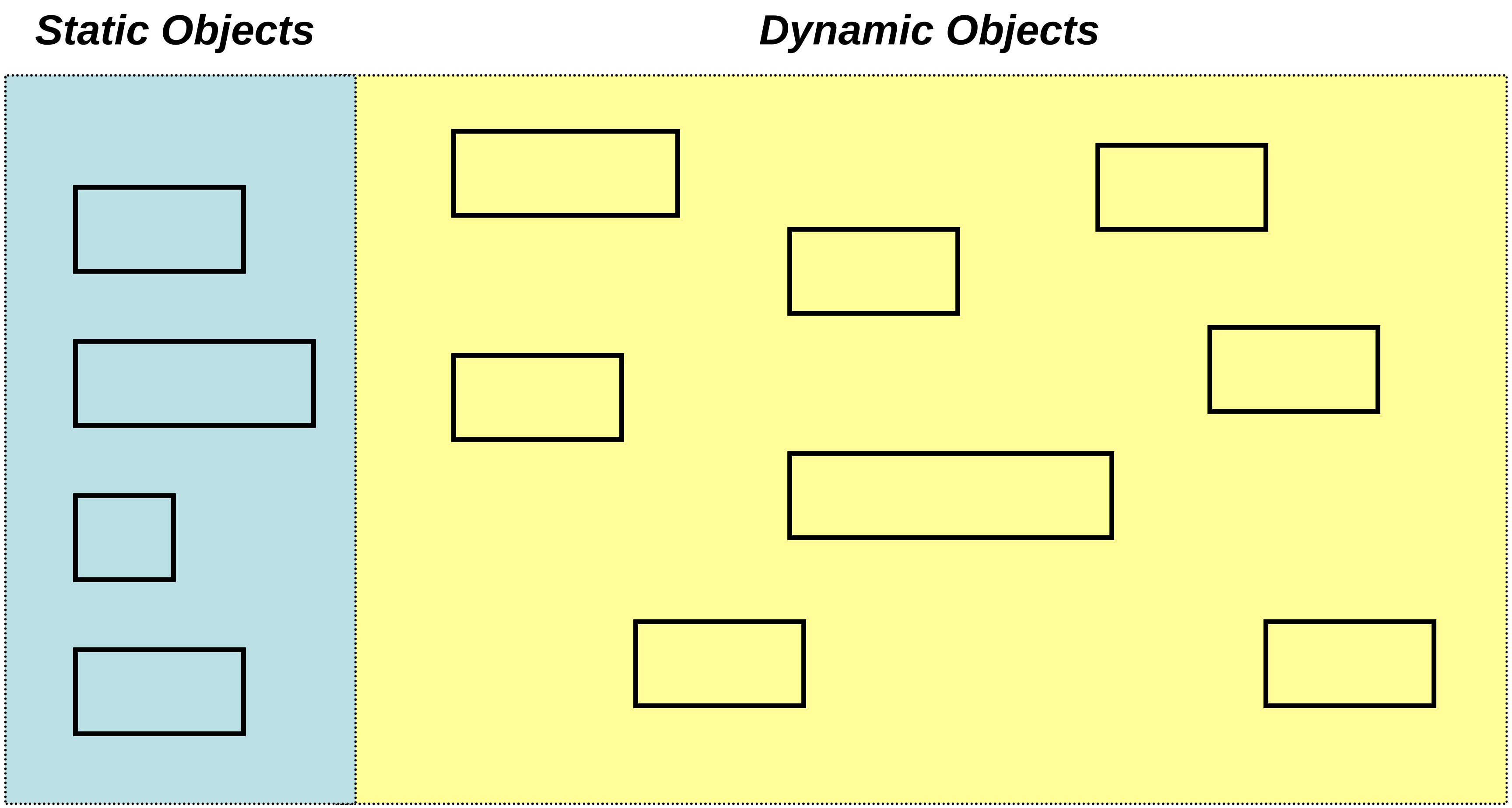
- Some objects are allocated dynamically

What about dynamic allocation?

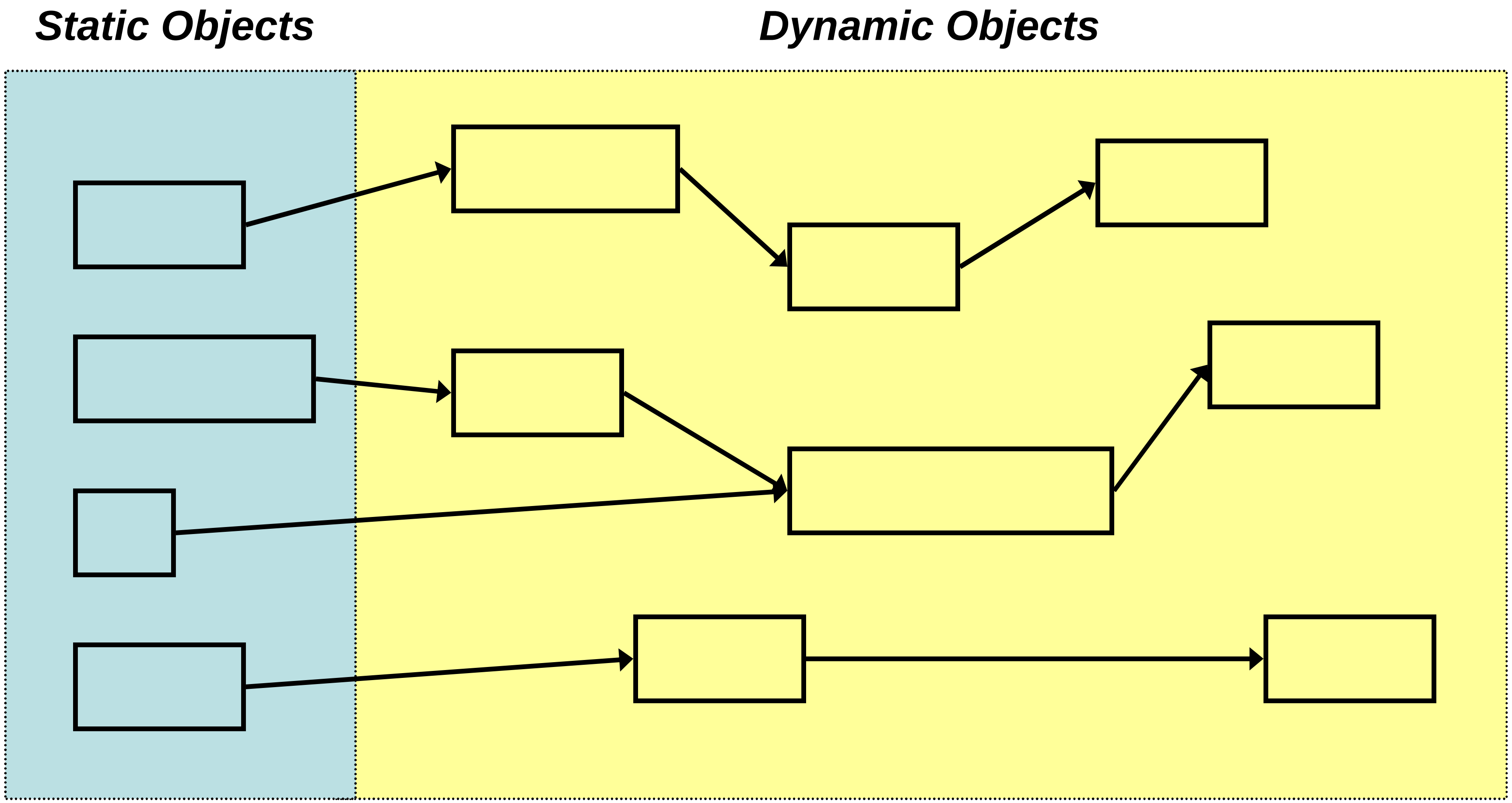
- Some objects are allocated dynamically



Simple observation #1



Simple observation #1



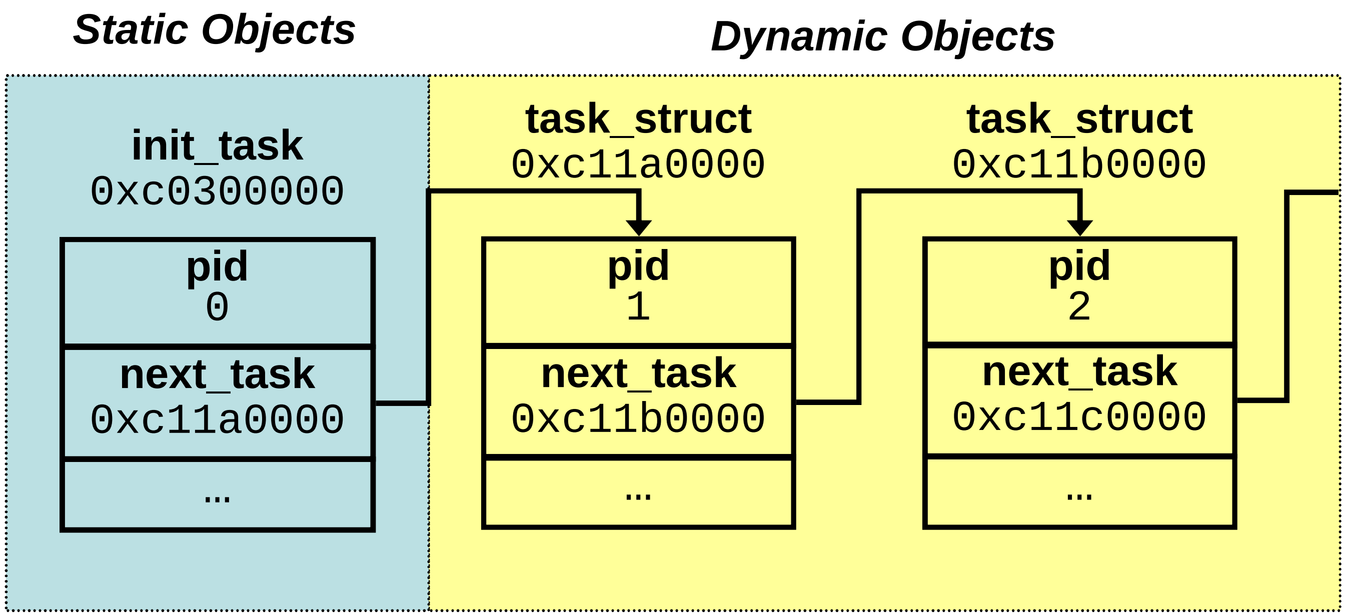
Simple observation #2

- The rootkit is just as ignorant as we are
- It will find dynamic objects by starting at static ones

“Combat tracking”

- Track rootkit **reads**
- Build a map of dynamic memory
- Reverse VMI

Combat tracking example

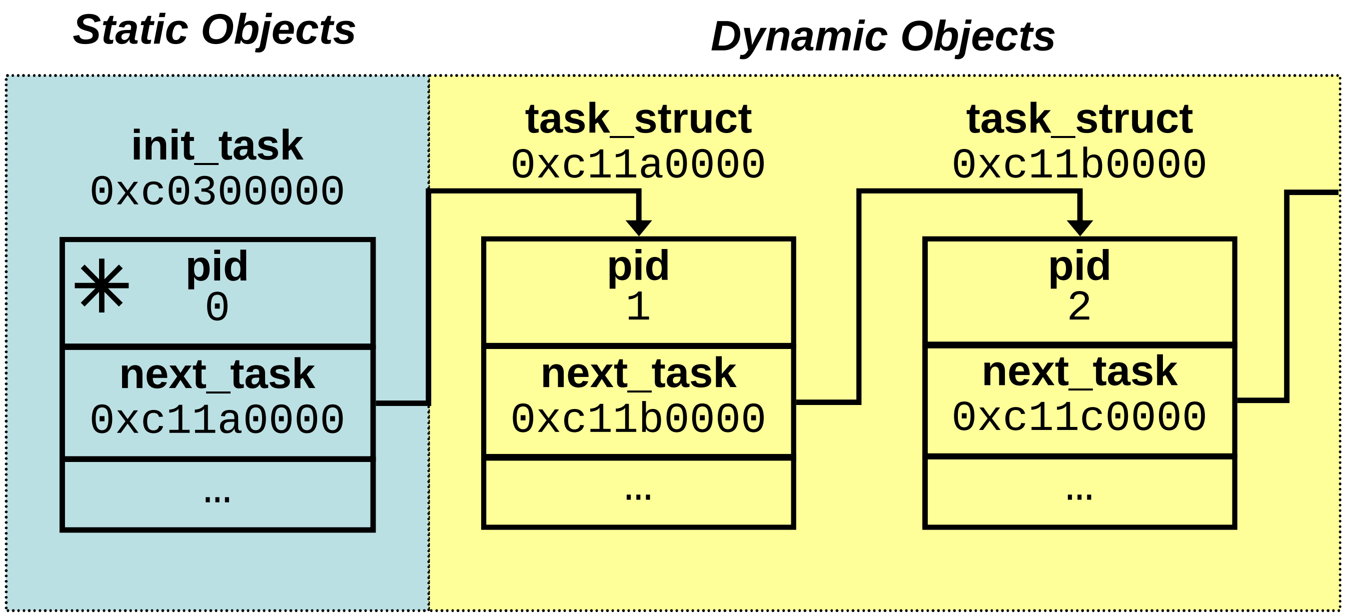


Memory Map

`0xc0300000` – task_struct

Output

Combat tracking example

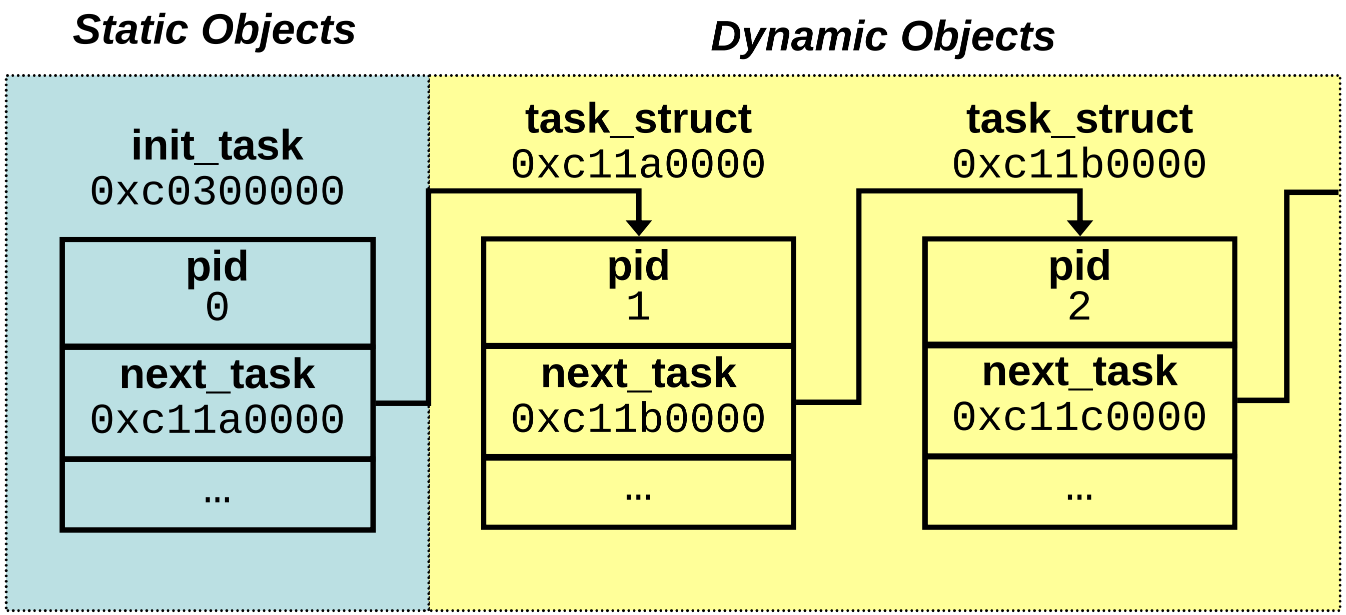


Memory Map

`0xc0300000` – task_struct

Output

Combat tracking example

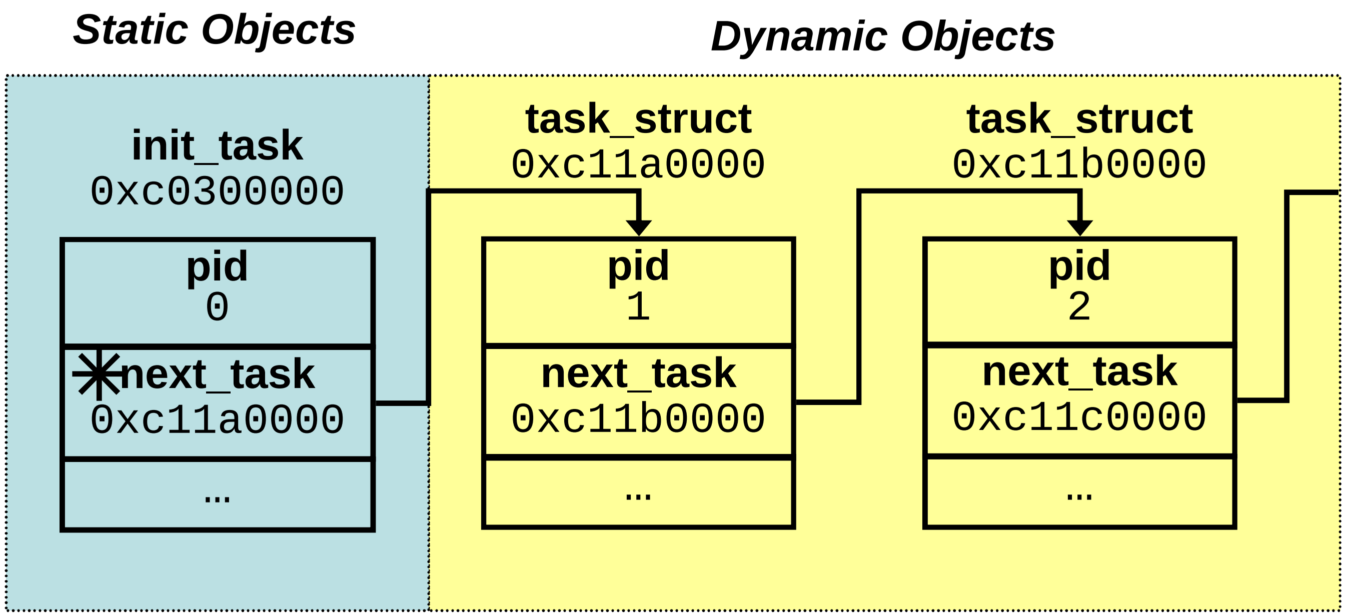


Memory Map

0xc0300000 – task_struct

Output

Combat tracking example

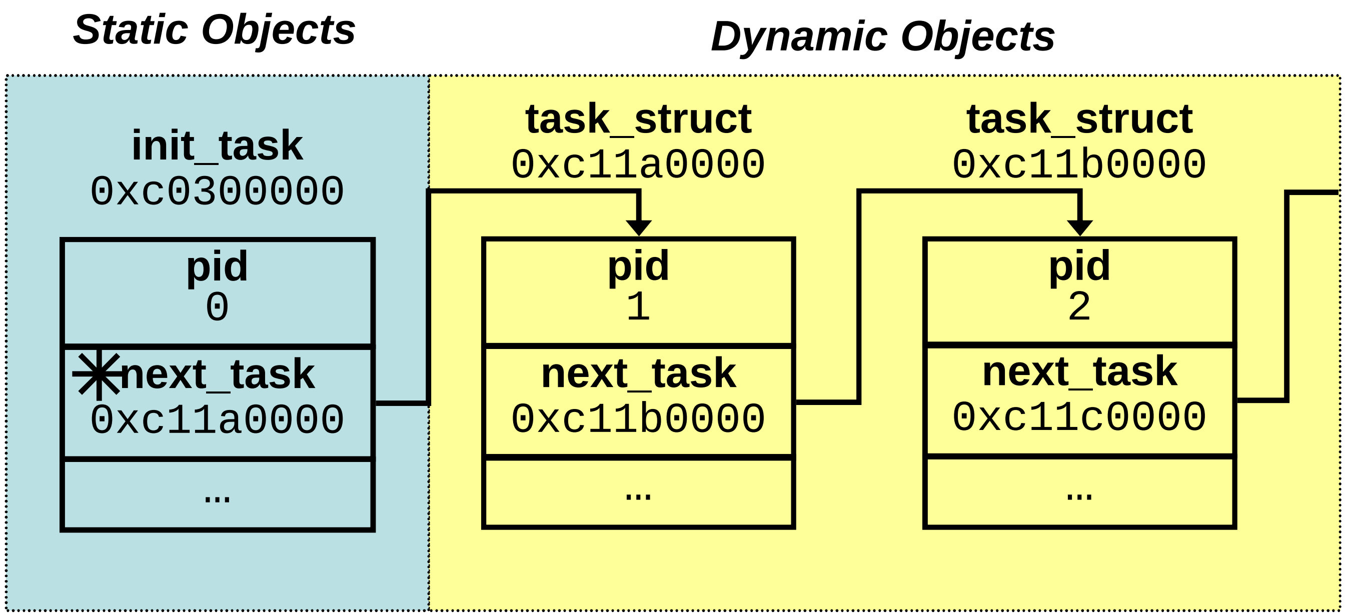


Memory Map

0xc0300000 – task_struct

Output

Combat tracking example



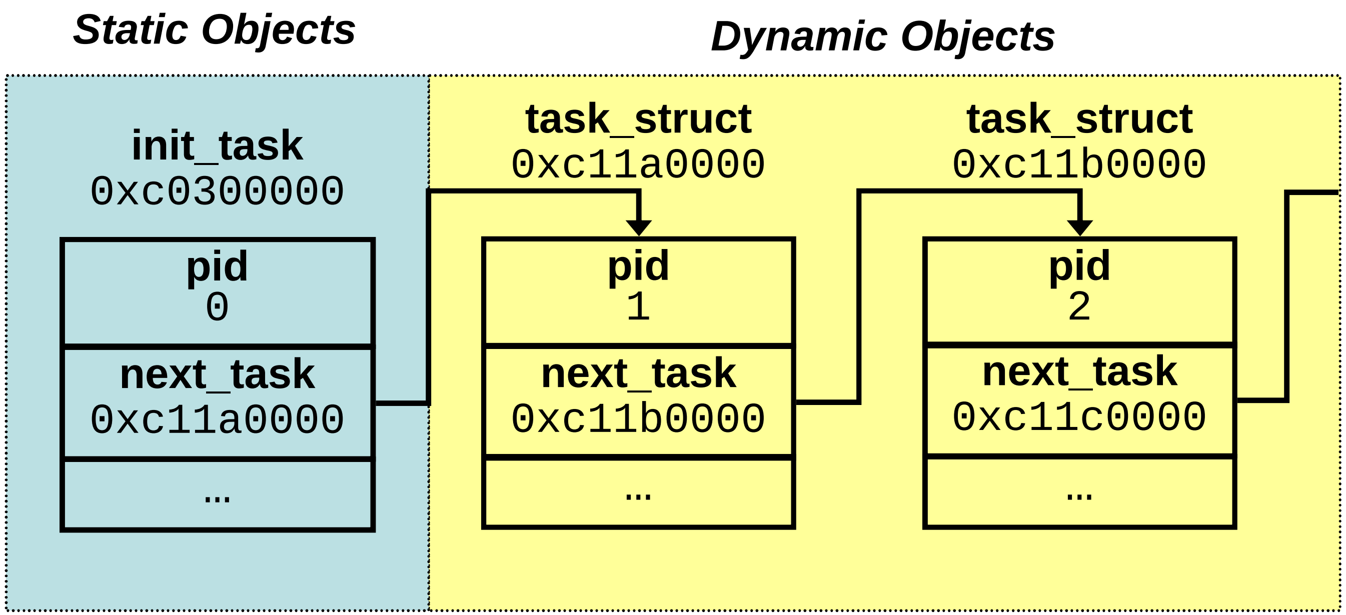
Memory Map

0xc0300000 – task_struct

0xc11a0000 – task_struct

Output

Combat tracking example



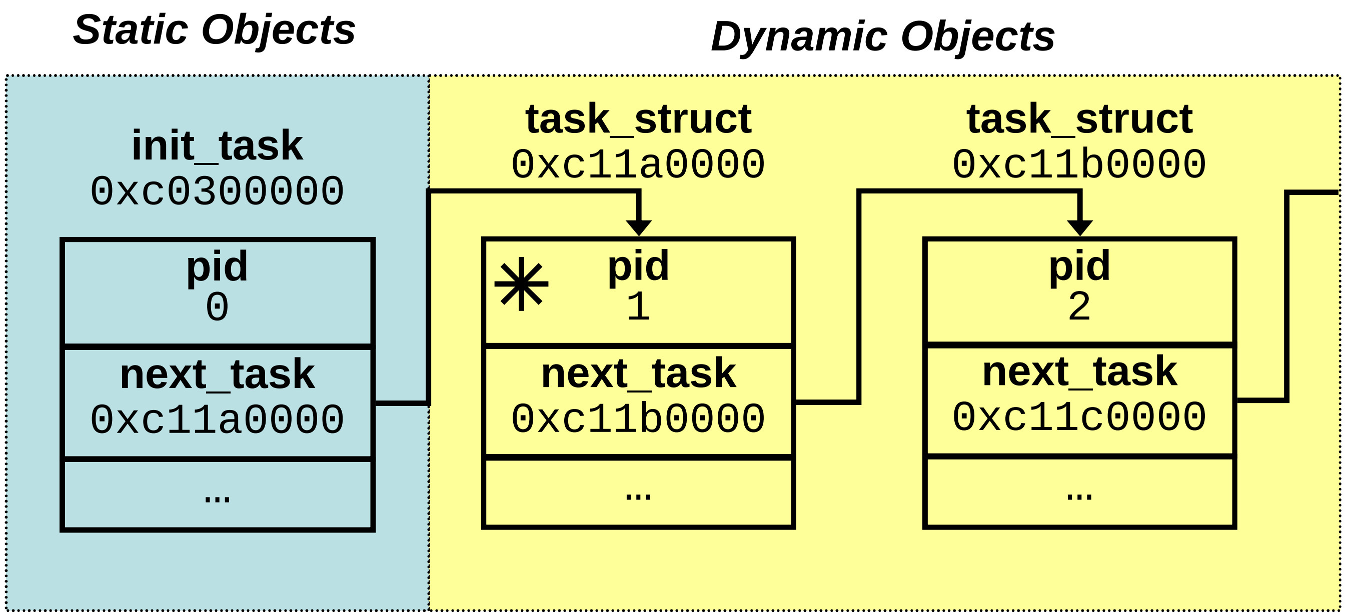
Memory Map

0xc0300000 – task_struct

0xc11a0000 – task_struct

Output

Combat tracking example



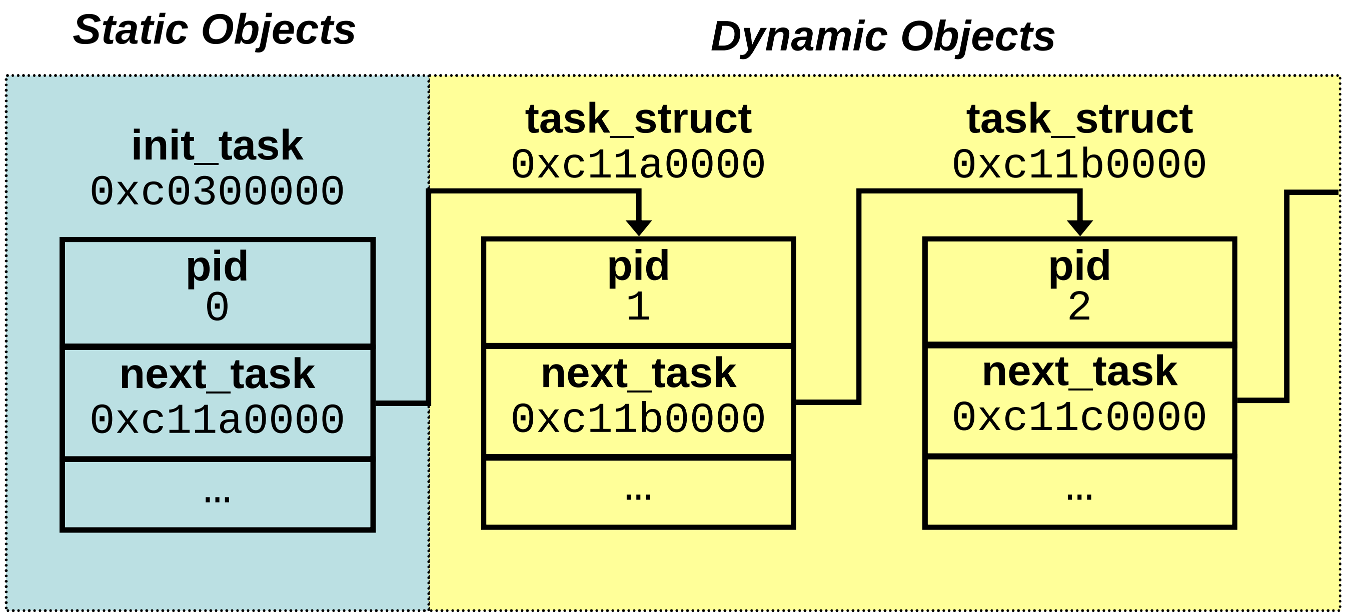
Memory Map

0xc0300000 – task_struct

0xc11a0000 – task_struct

Output

Combat tracking example



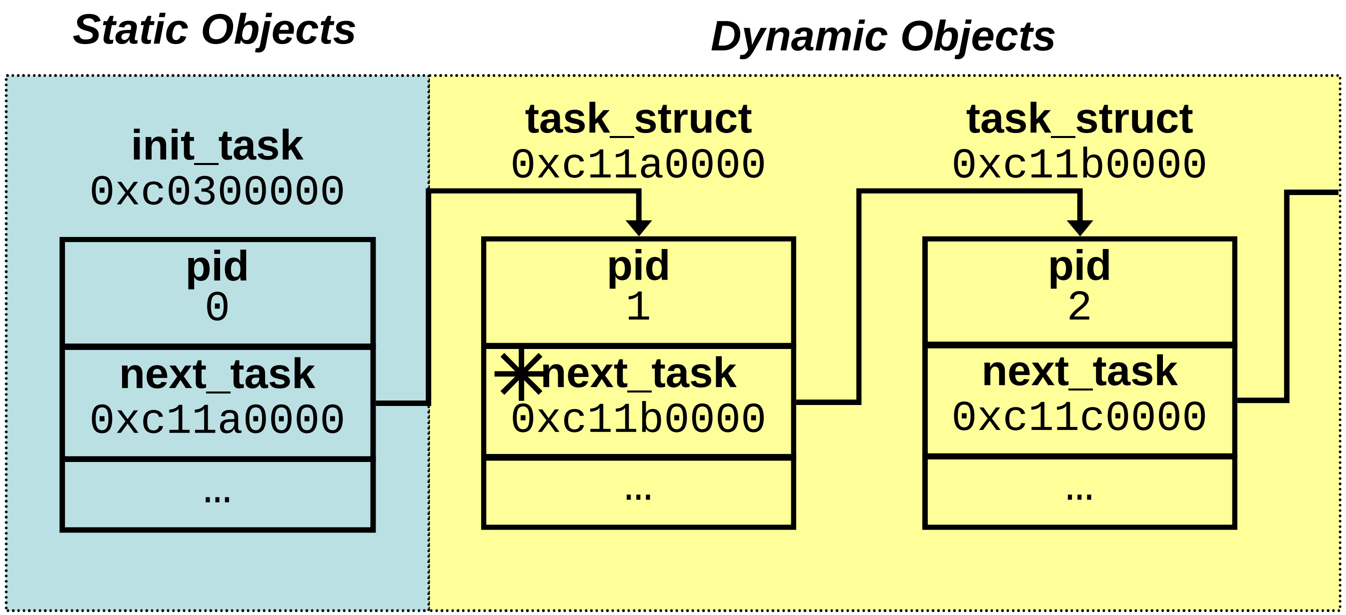
Memory Map

0xc0300000 – task_struct

0xc11a0000 – task_struct

Output

Combat tracking example



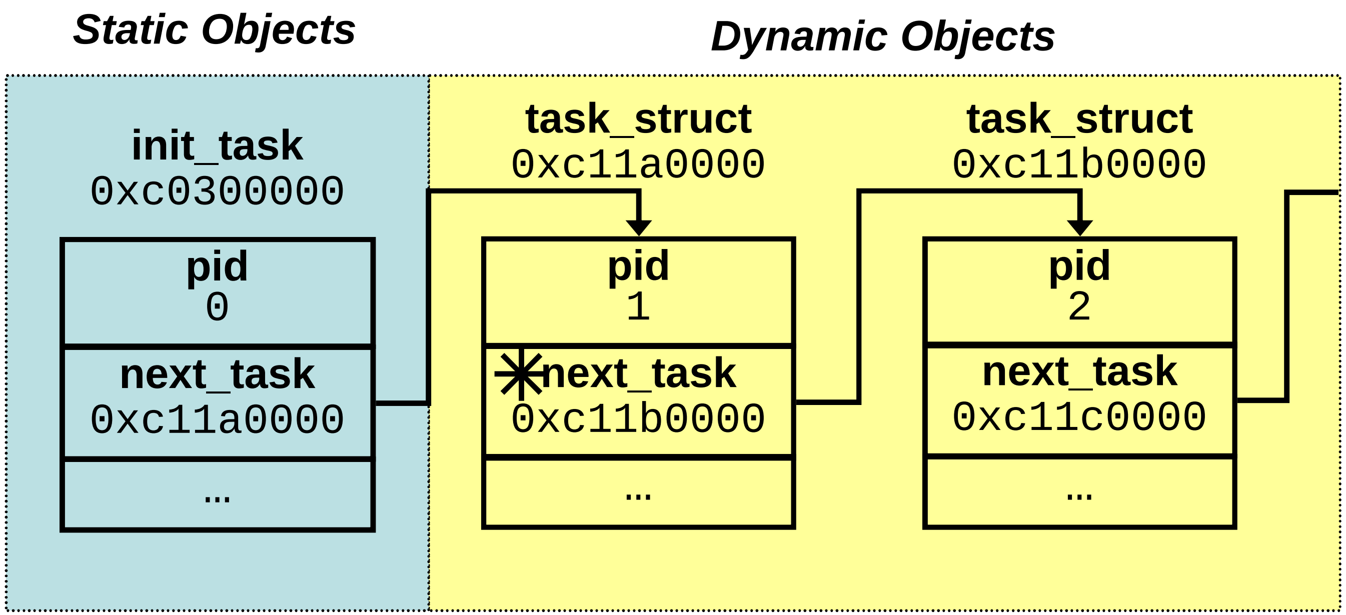
Memory Map

0xc0300000 – task_struct

0xc11a0000 – task_struct

Output

Combat tracking example



Memory Map

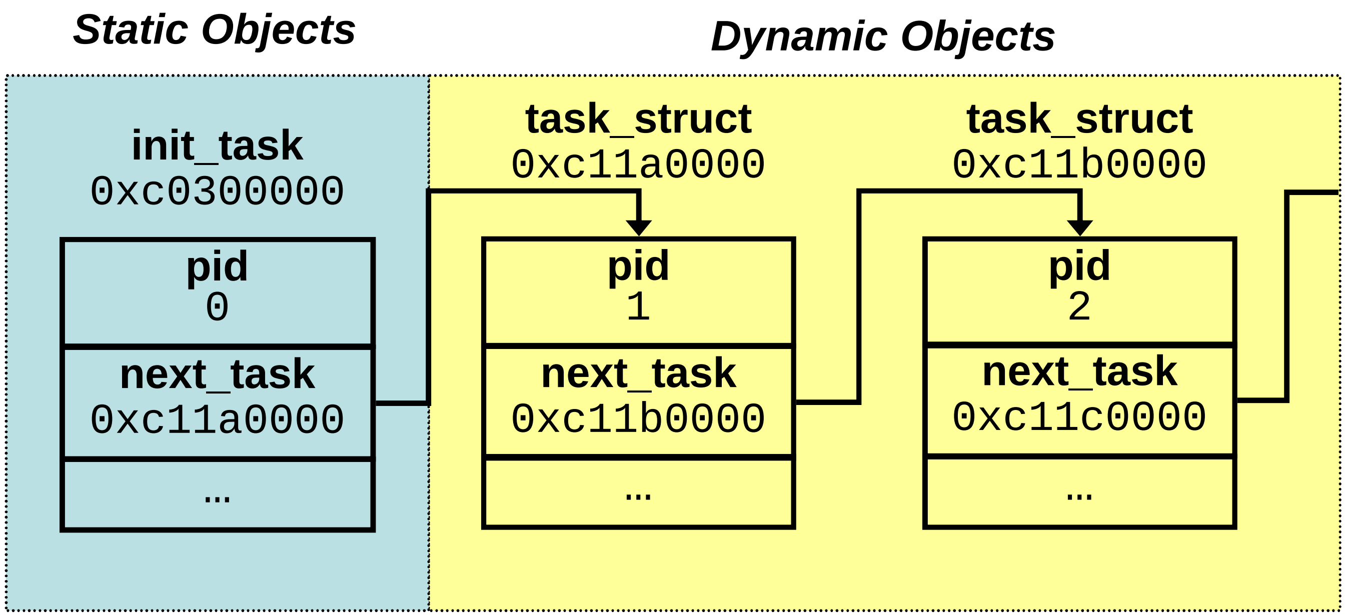
0xc0300000 – task_struct

0xc11a0000 – task_struct

0xc11b0000 – task_struct

Output

Combat tracking example



Memory Map

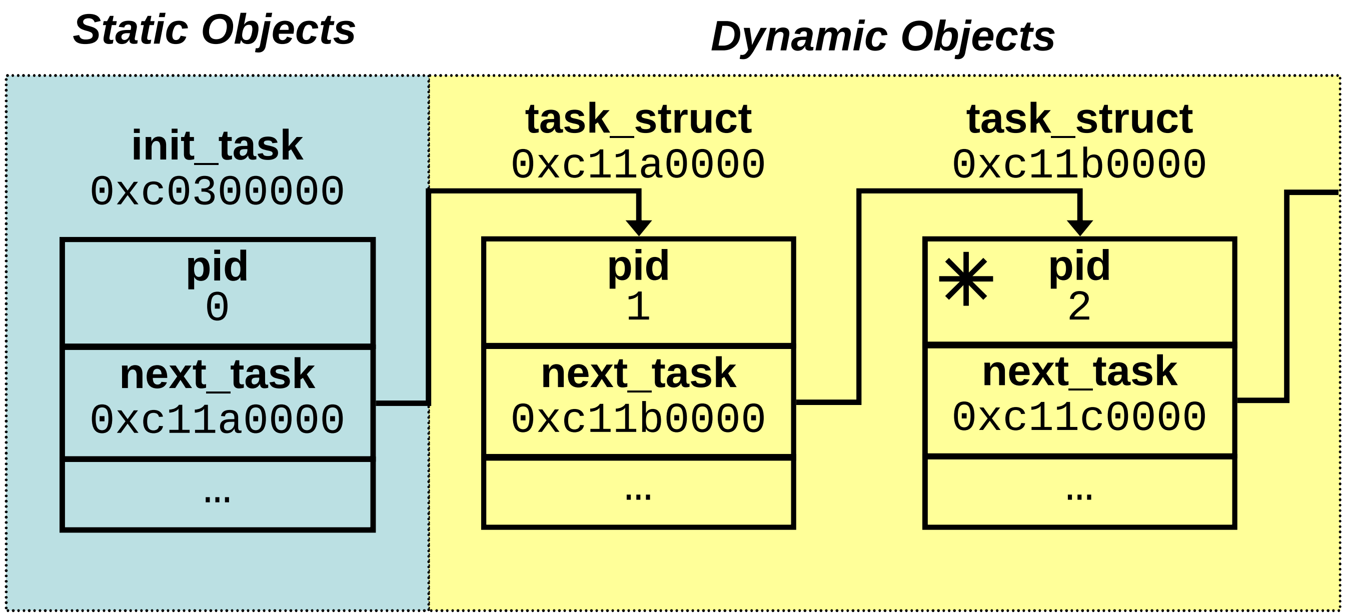
`0xc0300000` – task_struct

`0xc11a0000` – task_struct

`0xc11b0000` – task_struct

Output

Combat tracking example



Memory Map

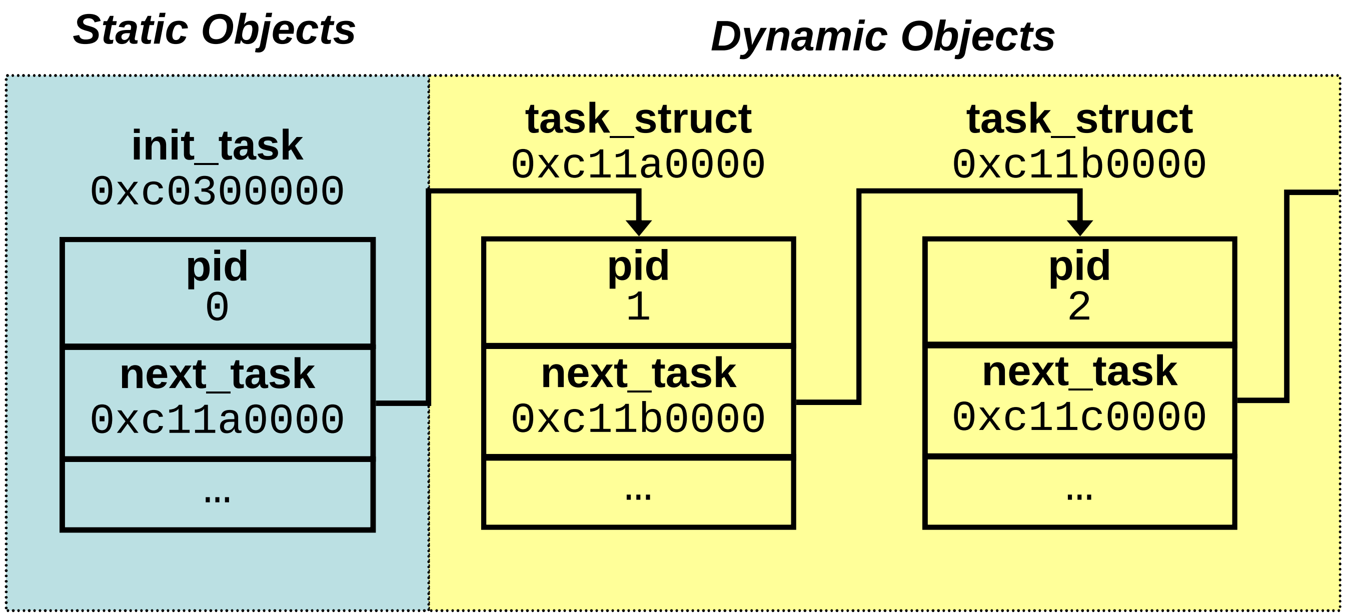
0xc0300000 – task_struct

0xc11a0000 – task_struct

0xc11b0000 – task_struct

Output

Combat tracking example



Memory Map

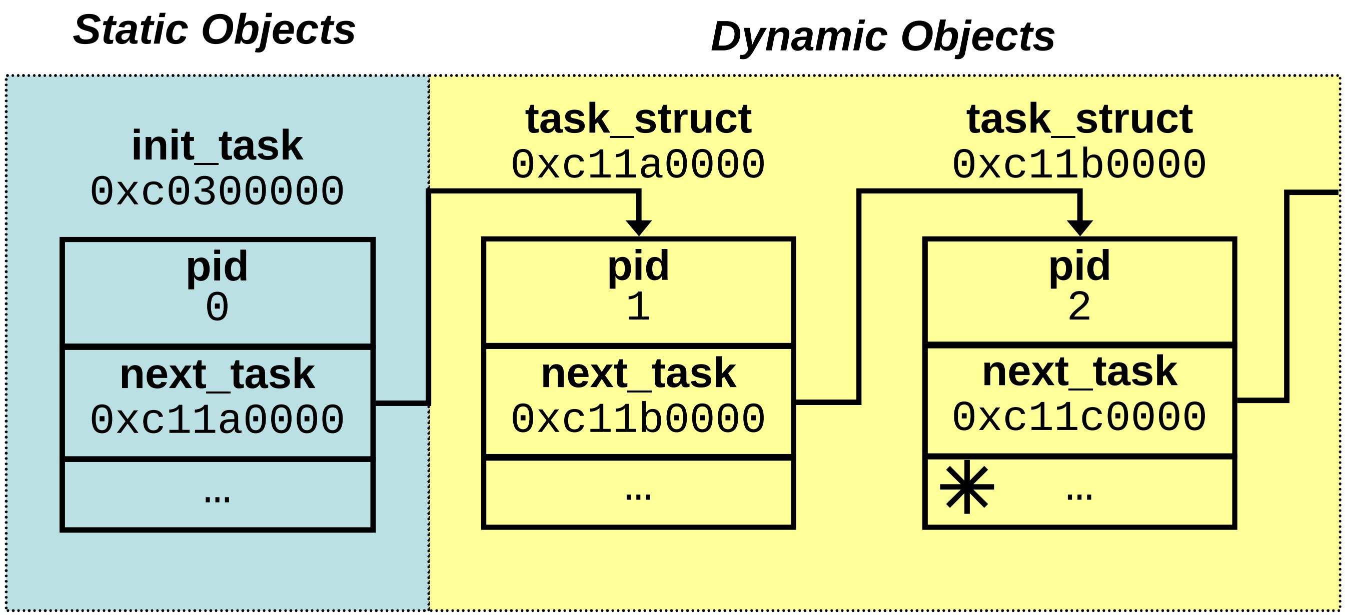
`0xc0300000` – task_struct

`0xc11a0000` – task_struct

`0xc11b0000` – task_struct

Output

Combat tracking example



Memory Map

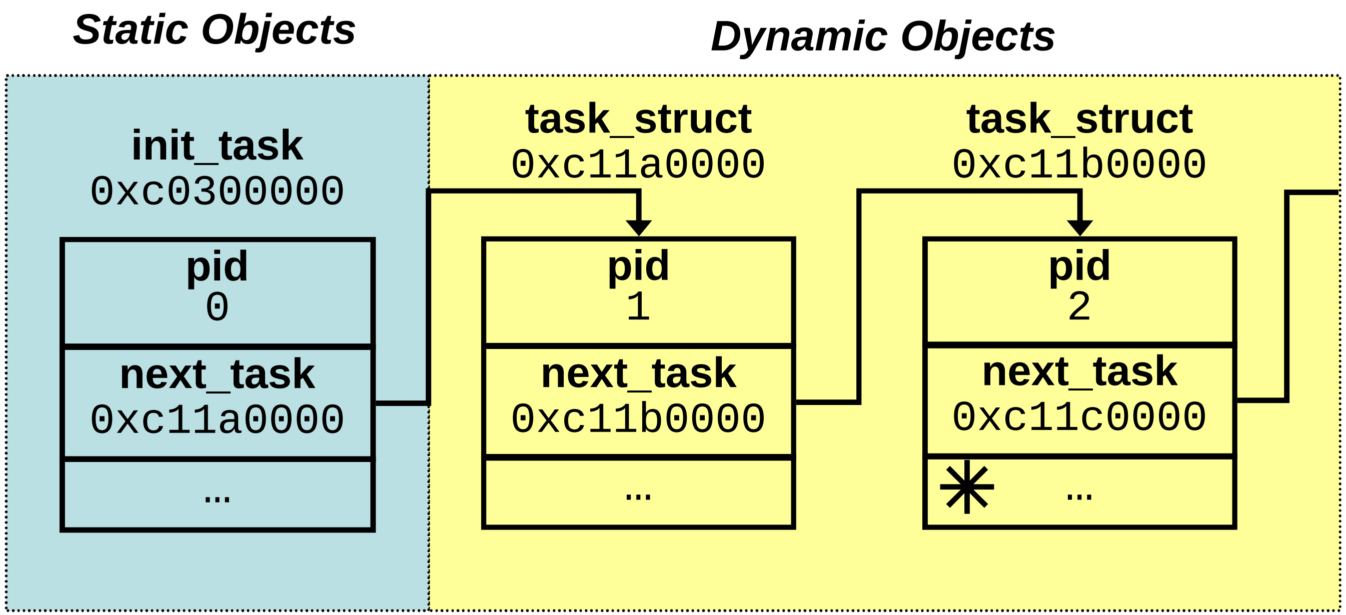
0xc0300000 – task_struct

0xc11a0000 – task_struct

0xc11b0000 – task_struct

Output

Combat tracking example



Memory Map

0xc0300000 – task_struct

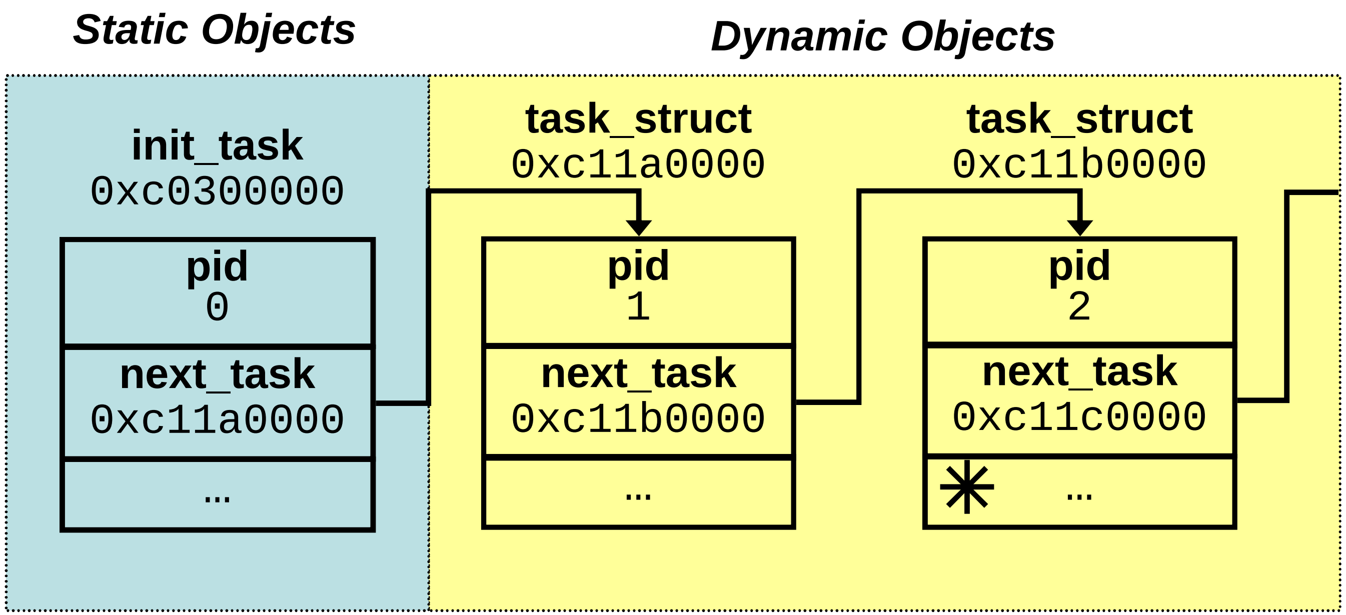
0xc11a0000 – task_struct

0xc11b0000 – task_struct

Output

Write to 0xc11b0056

Combat tracking example



Memory Map

`0xc0300000` – task_struct

`0xc11a0000` – task_struct

`0xc11b0000` – task_struct

Output

```
Write to 0xc11b0056
task_struct->euid
```

How does it modify control flow?

- Kernel hooks
  - Function pointers
  - Part of existing data objects
  - Could be statically or dynamically allocated
  - This is a subset of the previous point…

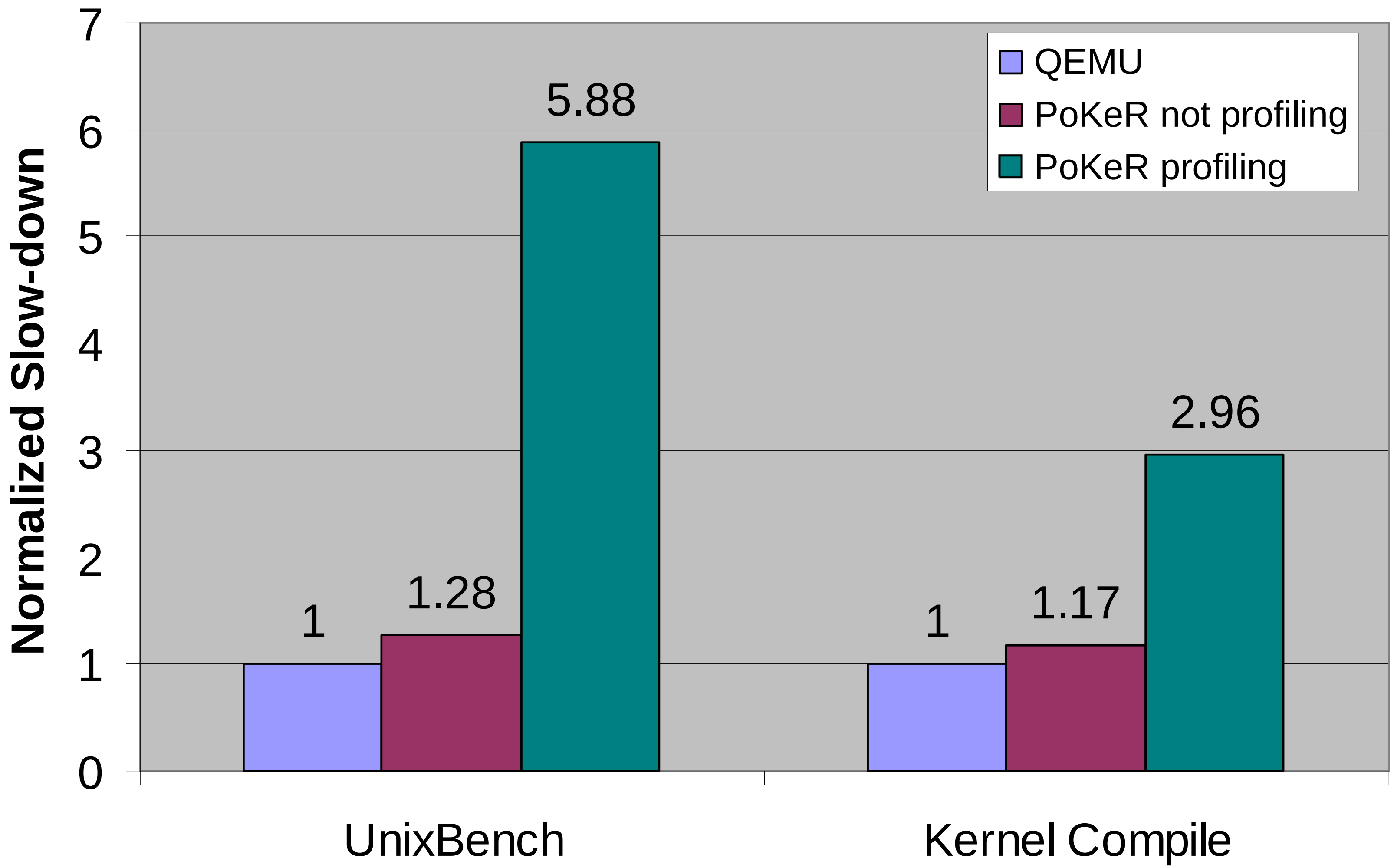
Results – adore

Name	Code	Kernel Objects Modified
adore 0.42	770 instr	sys_call_table[2,4,5,6,18,37,39,84,106] sys_call_table[107,120,141,195,196,220]
adore 0.53	733 instr	sys_call_table[1,2,6,26,37,39,120,141,220] proc_net->subdir->next->(...)->next->get_info proc_root_inode_operations->lookup
adore-ng 0.56	785 instr	proc_net->subdir->next->(...)->next->get_info proc_root_inode_operations->lookup proc_root_operations->readdir ext3_dir_operations->readdir ext3_file_operations->write unix_dgram_ops->recvmsg

Results – hp rootkit

Action	***Value***	***Kernel Object***
R	0xc677c000	pidhash[600]
R	0x0000025a	pidhash[600]->pid
R	0xc76d8000	pidhash[600]->next_task
R	0xc6780000	pidhash[600]->prev_task
W	**0xc6780000**	**pidhash[600]->next_task->prev_task**
W	**0xc76d8000**	**pidhash[600]->prev_task->next_task**

Performance



Limitations

- Lack of formal completeness
- Cannot reveal the reason for modifications
- Combat tracking evasion
- Assume VMM isolation
- Kernel rootkits only

Related work

Panorama
CCS ‘07

HookFinder
NDSS ‘08

HookMap
RAID ’08

K-Tracer
NDSS ‘09

Your three take aways…

- PoKeR: Virtualization based rootkit profiler

- Combat Tracking allows us to track dynamic data objects

- Tells *what* a rootkit does in order to help an expert determine *why* it does it