Stony Brook University

Virtual Machine Introspection

Bhushan Jain

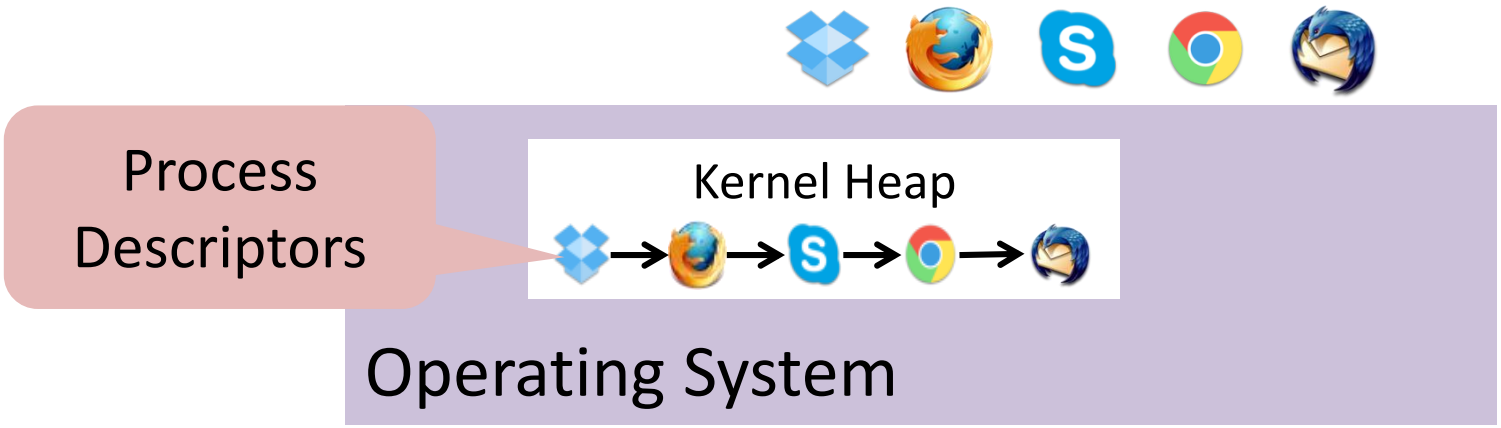
Computer Science Department

Stony Brook University

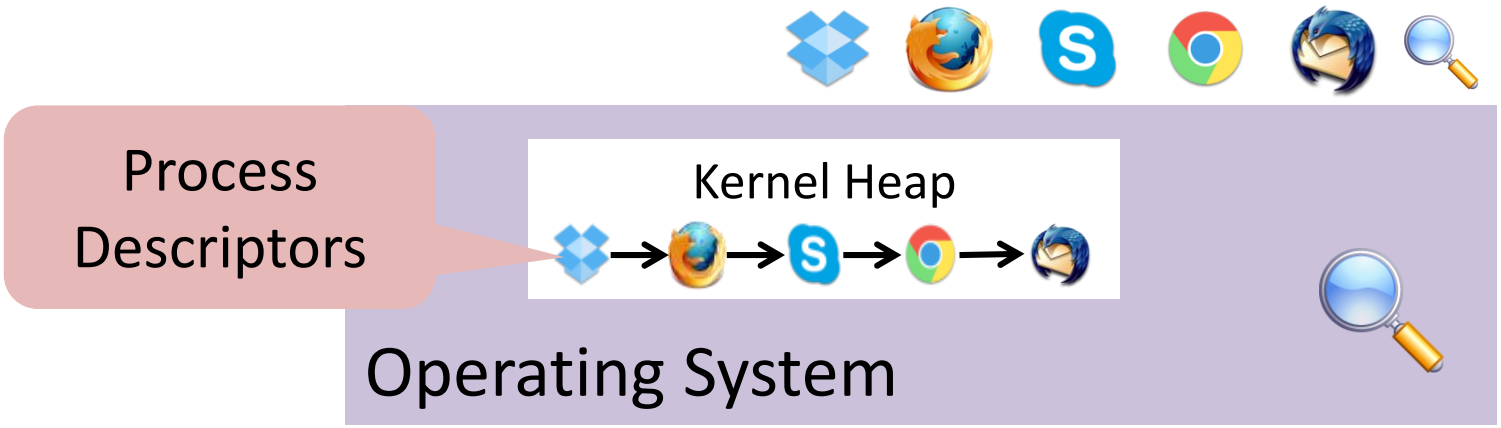
Traditional Environment

Operating System

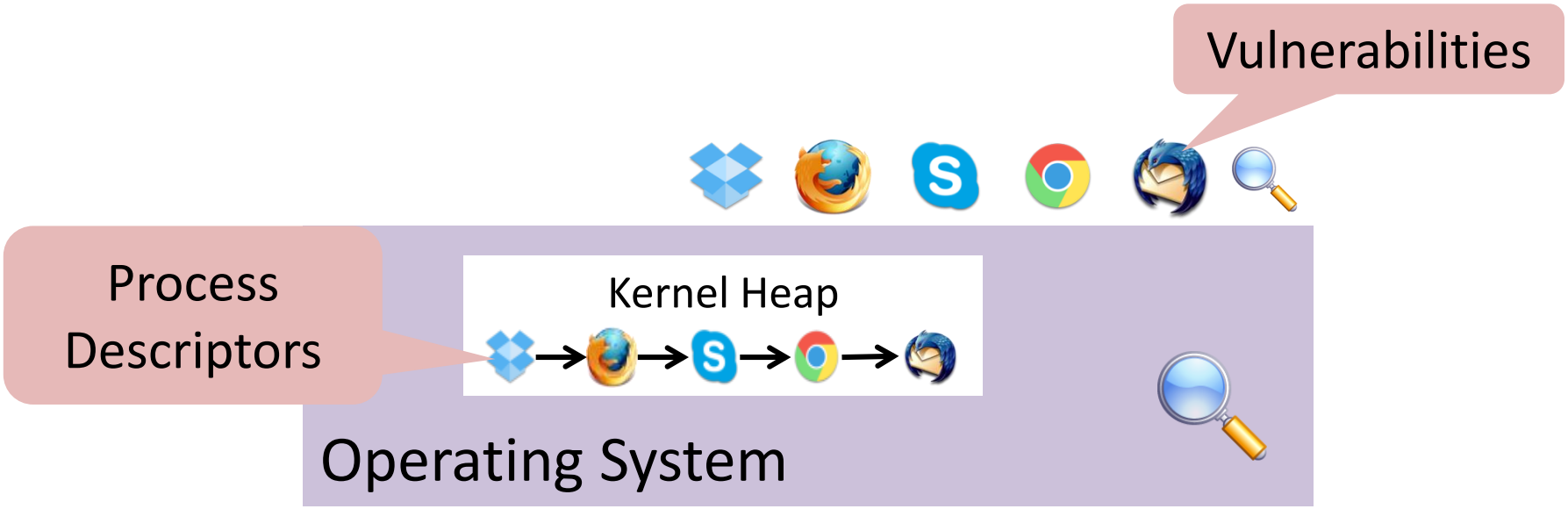
Traditional Environment



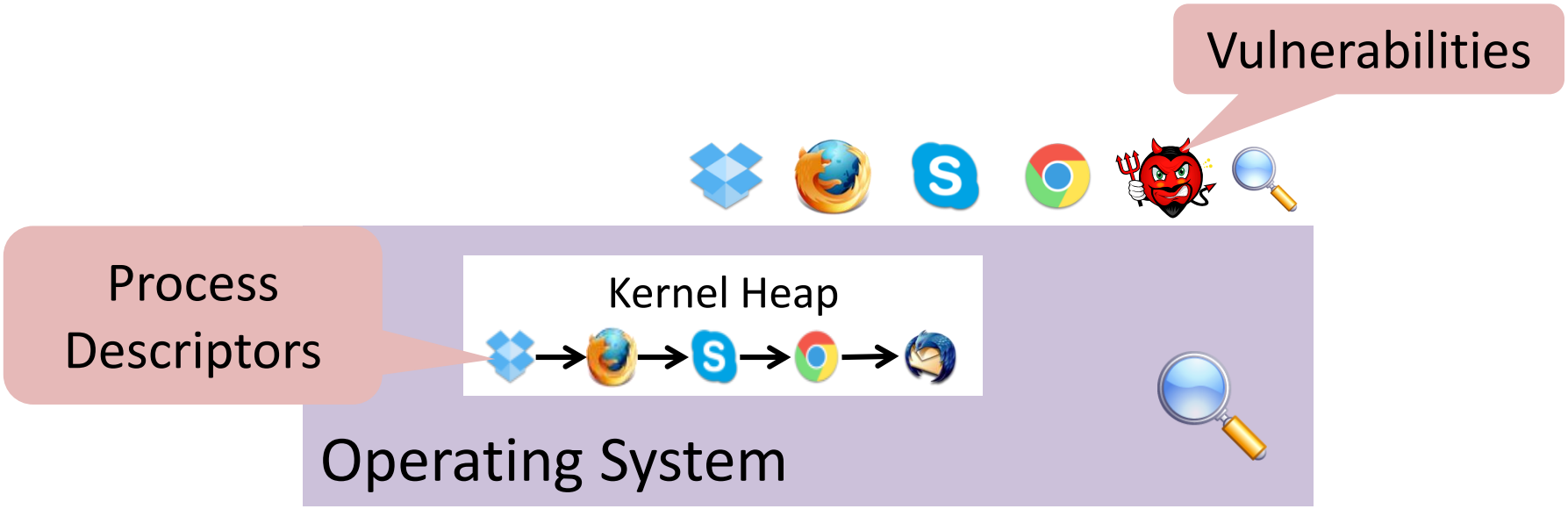
Traditional Environment



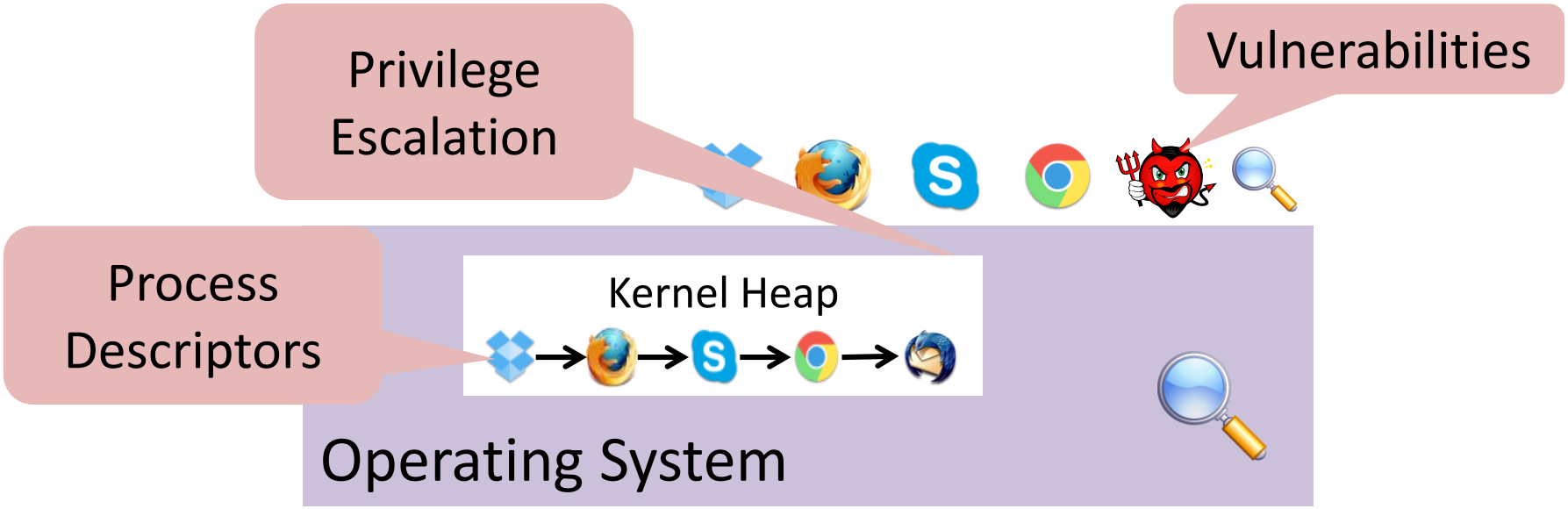
Traditional Environment



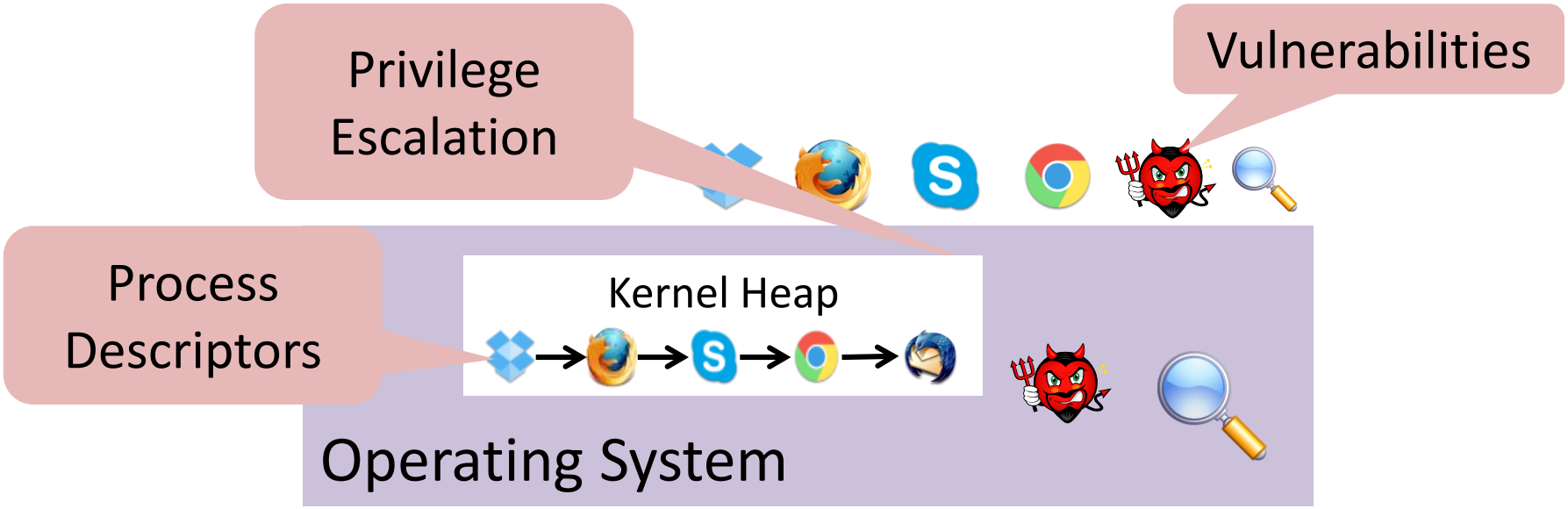
Traditional Environment



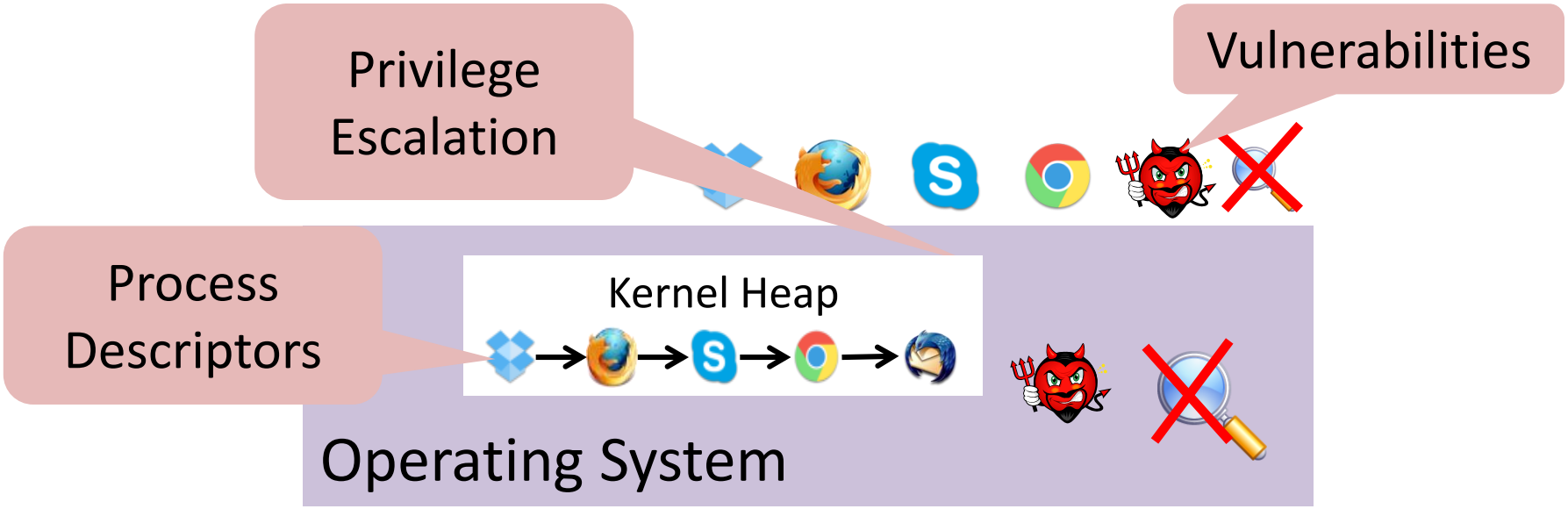
Traditional Environment



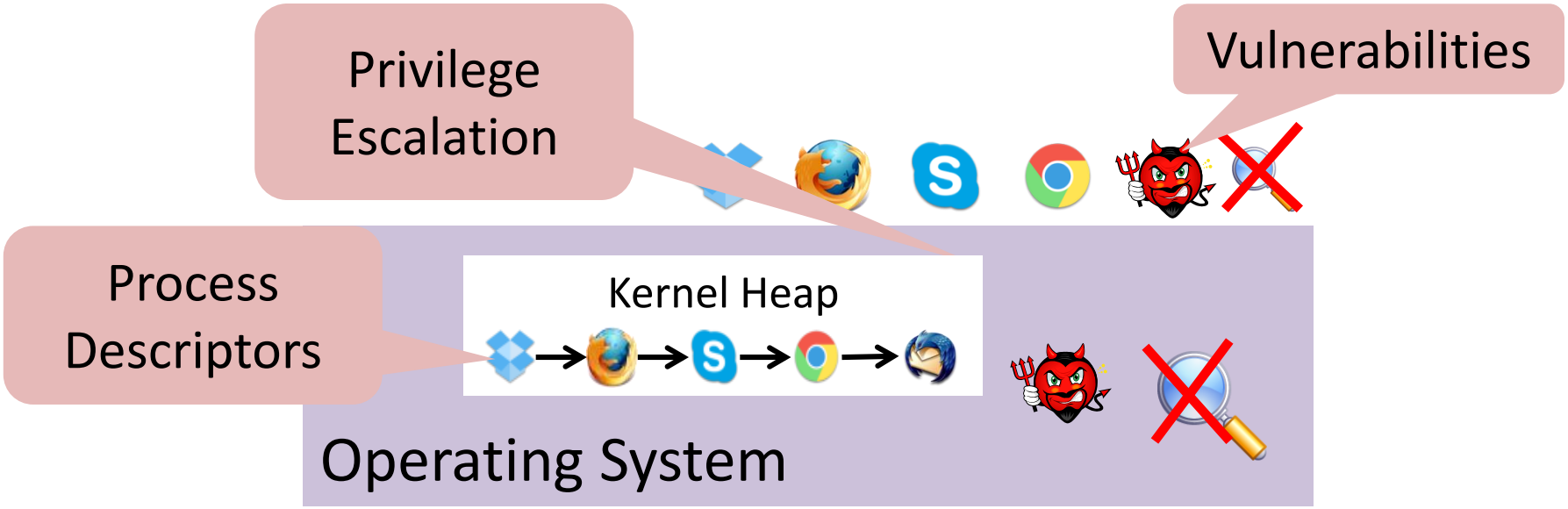
Traditional Environment



Traditional Environment

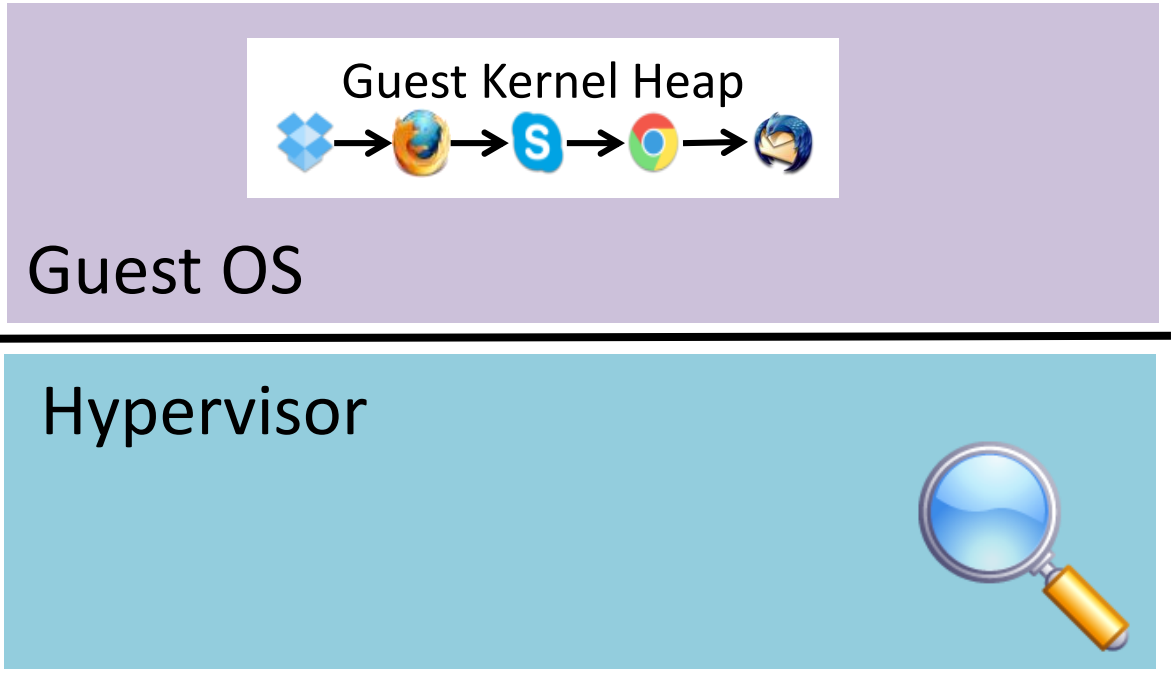


Traditional Environment

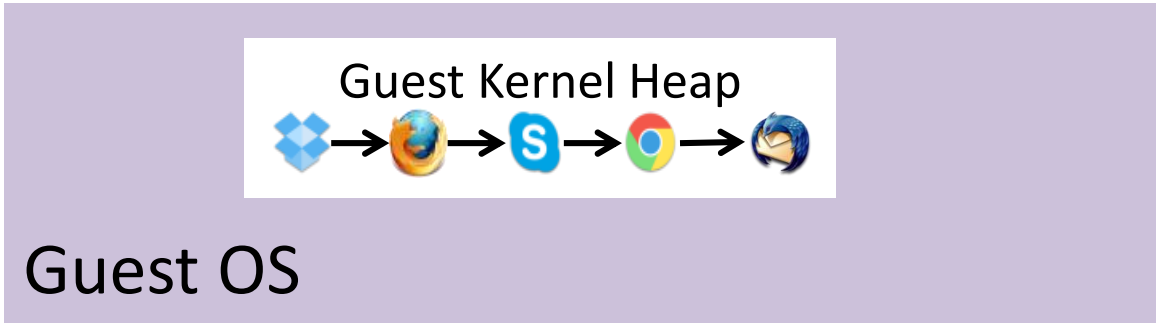


Security monitors can be easily subverted by rootkits

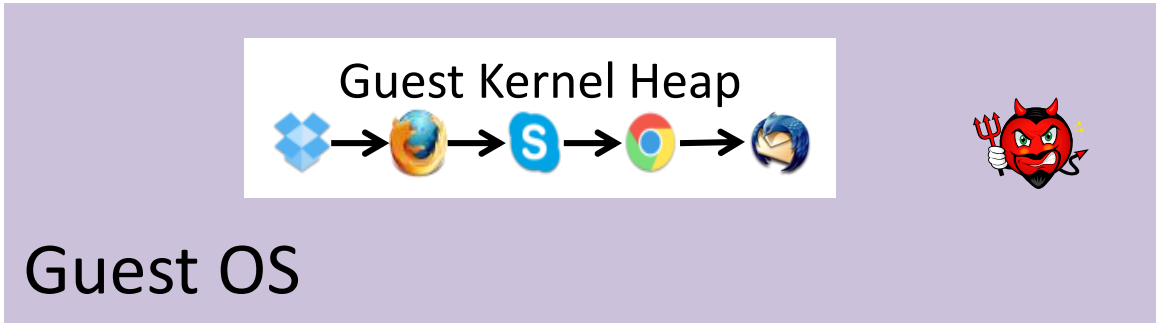
Layered Security



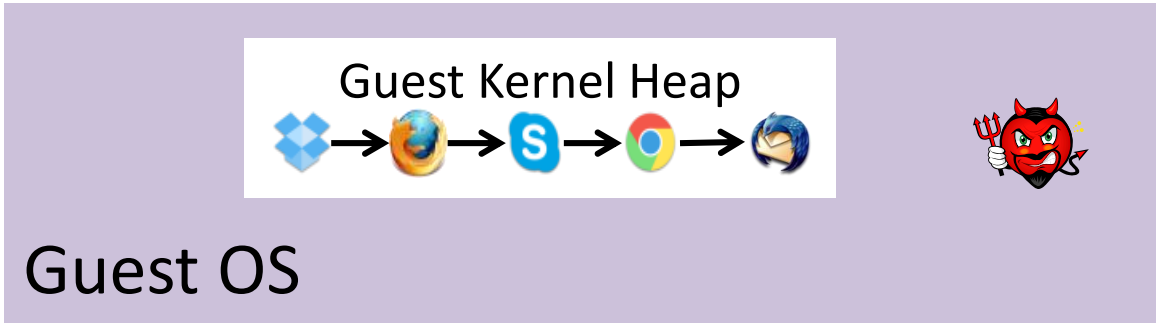
Layered Security



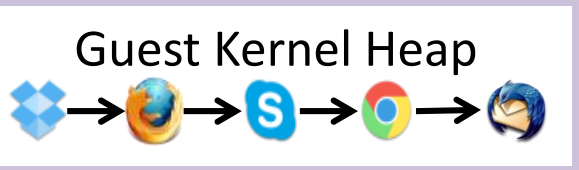
Layered Security



Layered Security



Layered Security

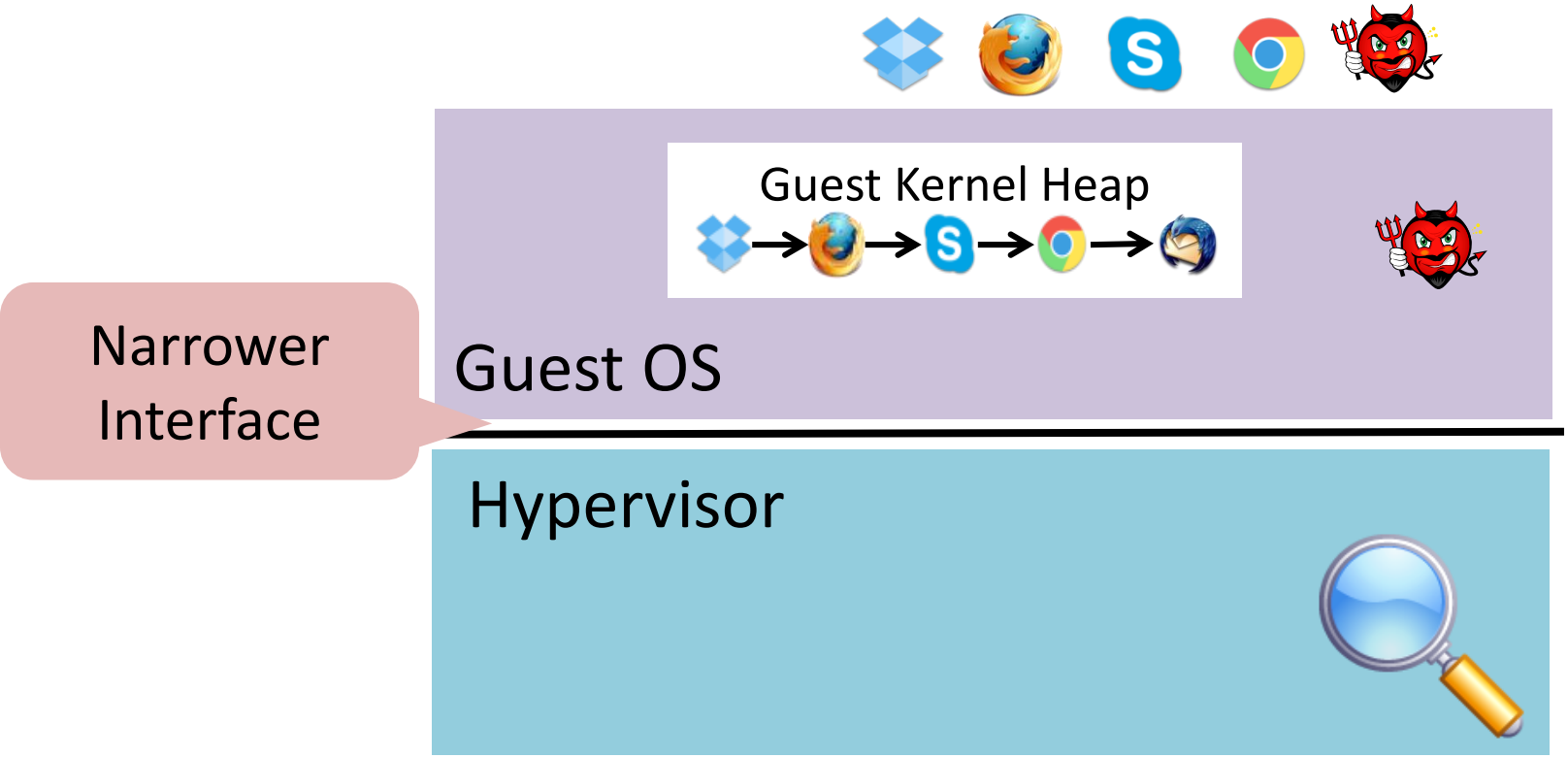


Guest OS

Narrower Interface

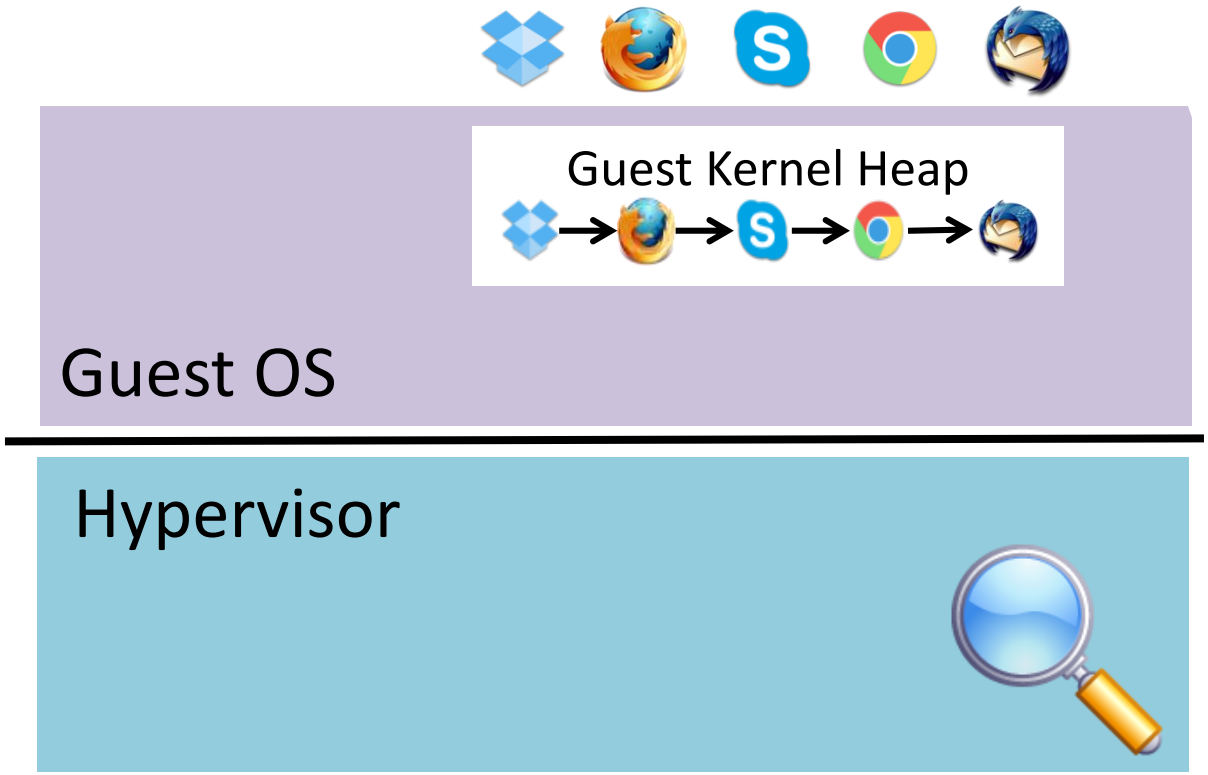
Hypervisor

Layered Security

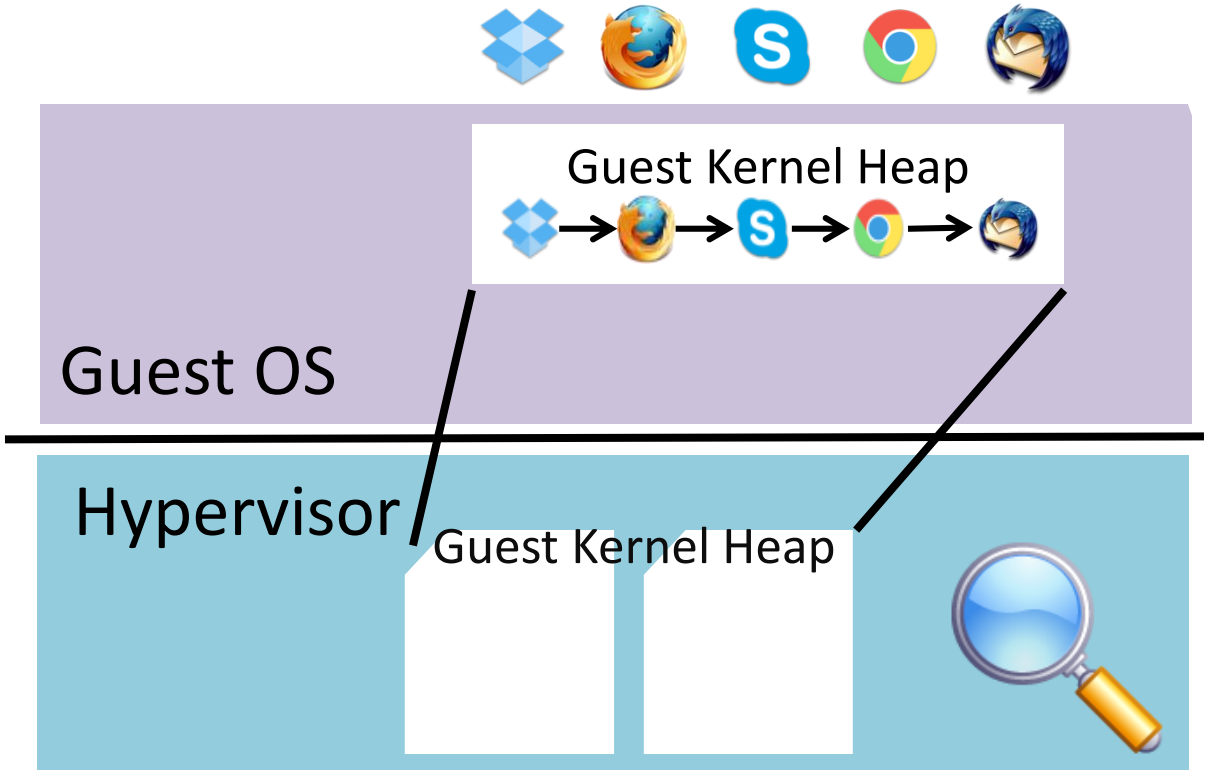


Narrower Interface = Fewer exploitable vulnerabilities

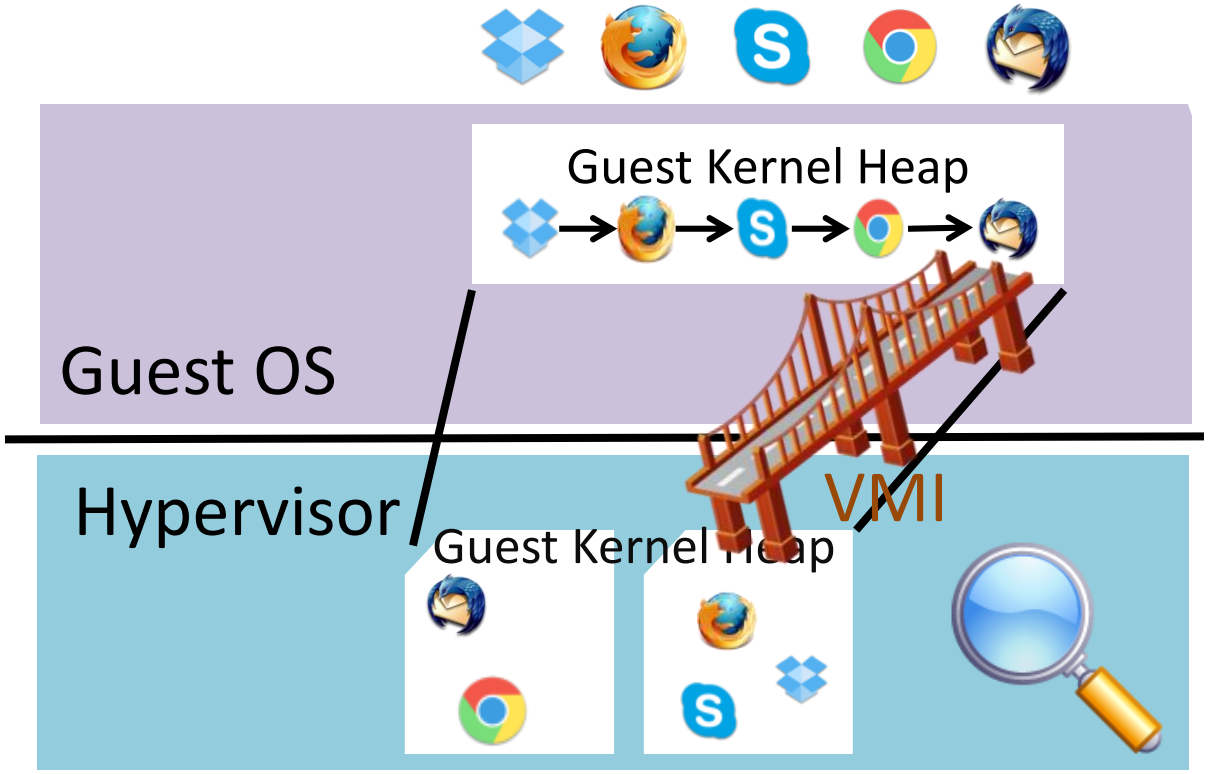
Virtual Machine Introspection



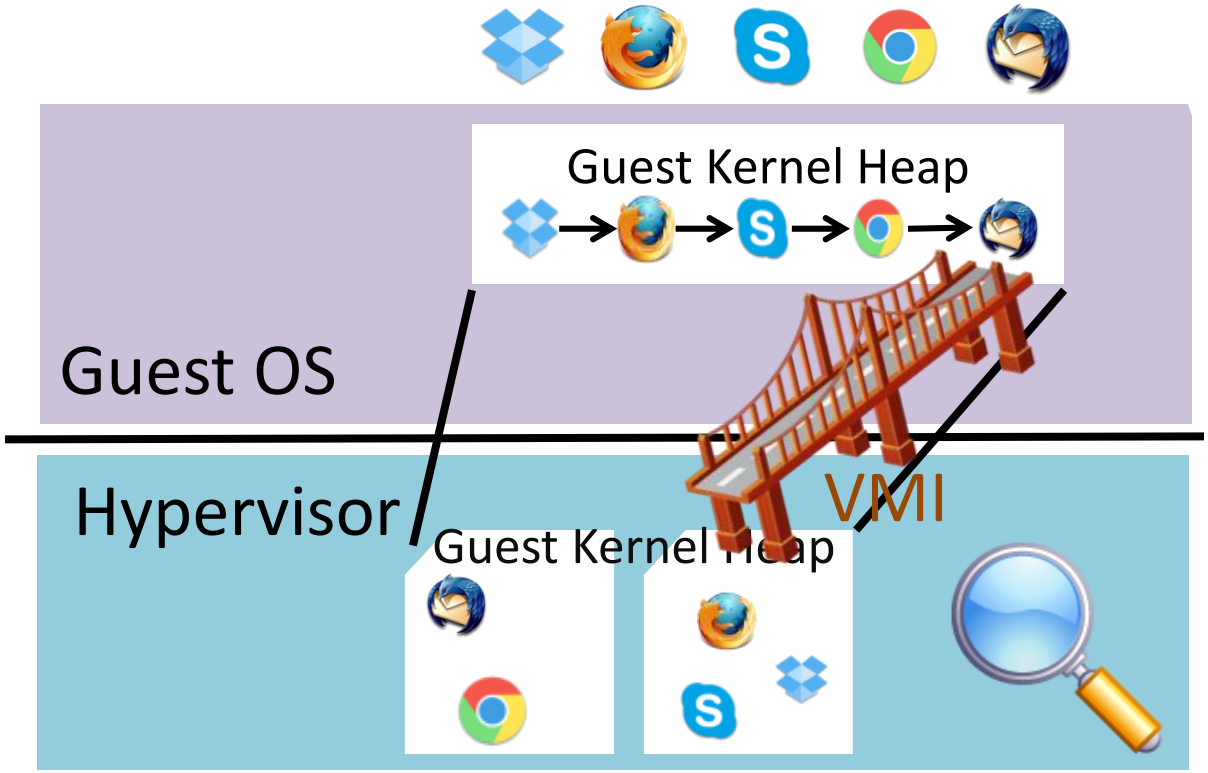
Virtual Machine Introspection



Virtual Machine Introspection



Virtual Machine Introspection



VMI: A technique to monitor the guest activities from VMM

Applications of VMI

- Introspect VM memory and CPU registers
  -
- Introspect disk contents
  -
- Network traffic
  -

Applications of VMI

- Introspect VM memory and CPU registers
  - E.g., List all running processes, open sockets, open files
- Introspect disk contents
  -
- Network traffic
  -

Applications of VMI

- Introspect VM memory and CPU registers
  - E.g., List all running processes, open sockets, open files
- Introspect disk contents
  - E.g., Differentiate VM data vs metadata
- Network traffic
  -

Applications of VMI

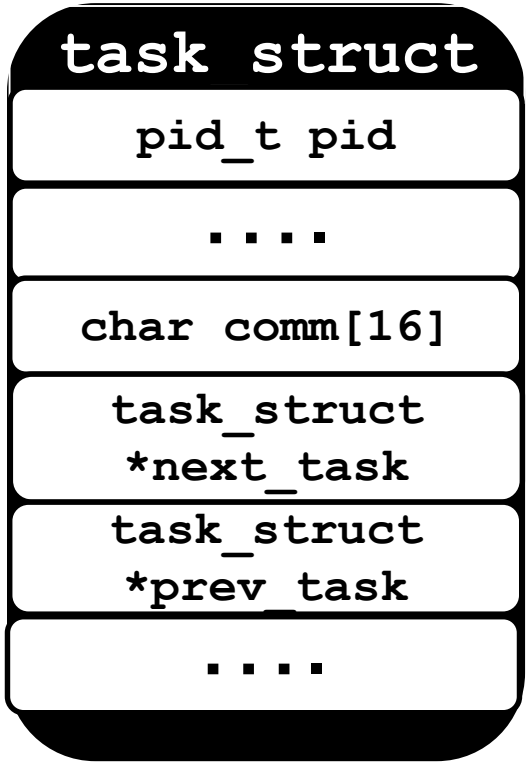
- Introspect VM memory and CPU registers
  - E.g., List all running processes, open sockets, open files
- Introspect disk contents
  - E.g., Differentiate VM data vs metadata
- Network traffic
  - E.g., Intrusion Prevention Systems

Applications of VMI

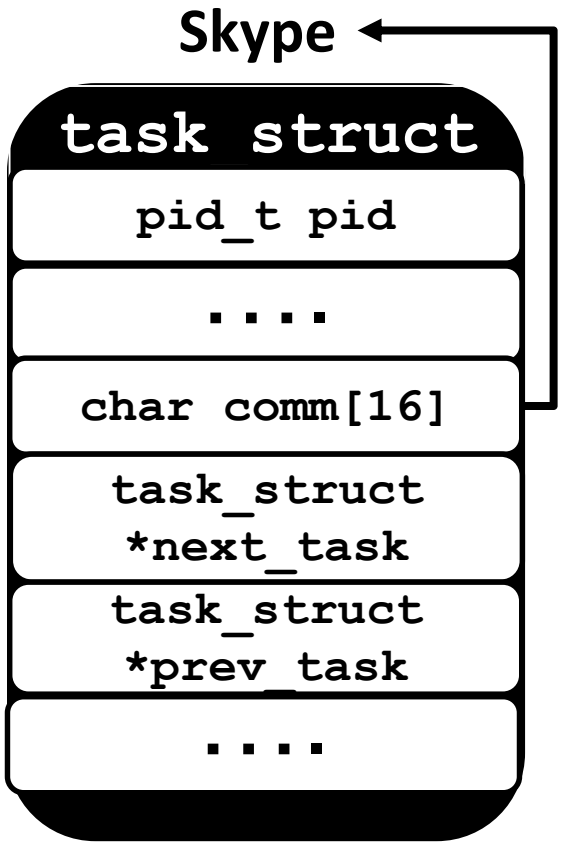
- Introspect VM memory and CPU registers
  - E.g., List all running processes, open sockets, open files
- Introspect disk contents
  - E.g., Differentiate VM data vs metadata
- Network traffic
  - E.g., Intrusion Prevention Systems

VMI is useful for more than just monitoring guest OS

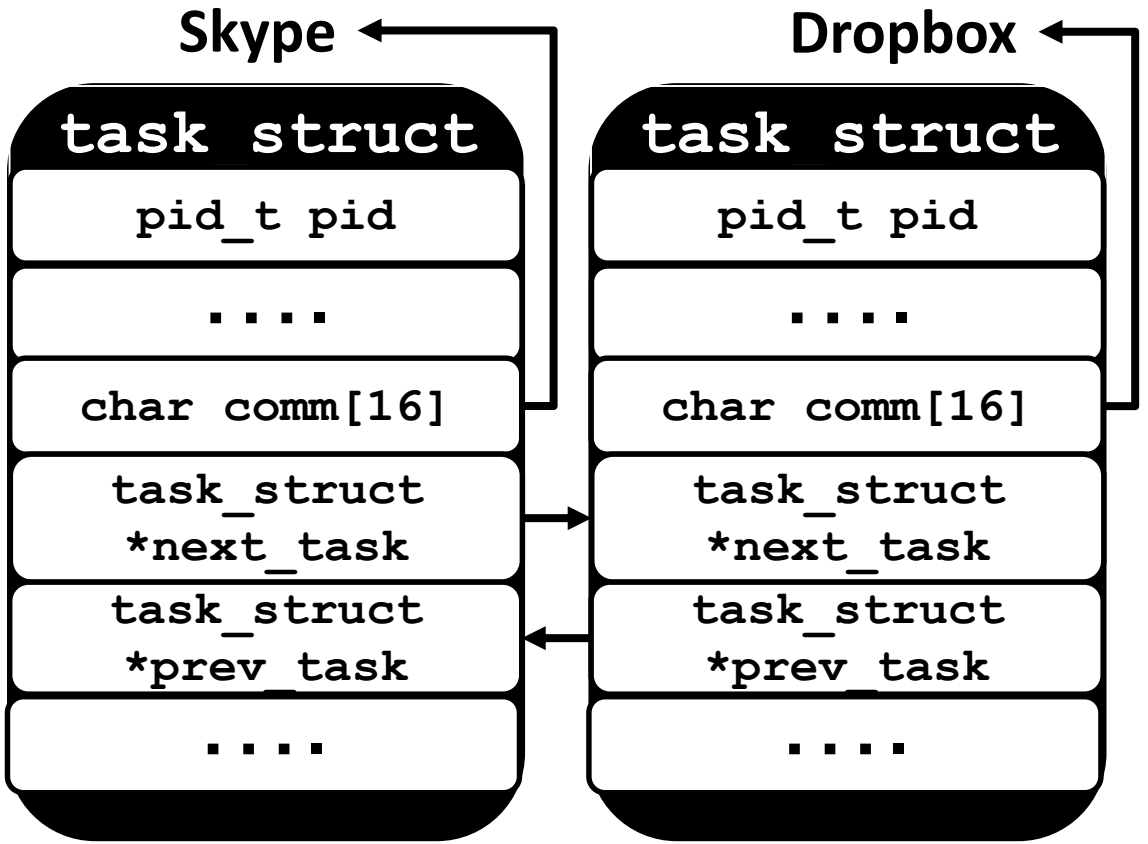
VMI In Action



VMI In Action

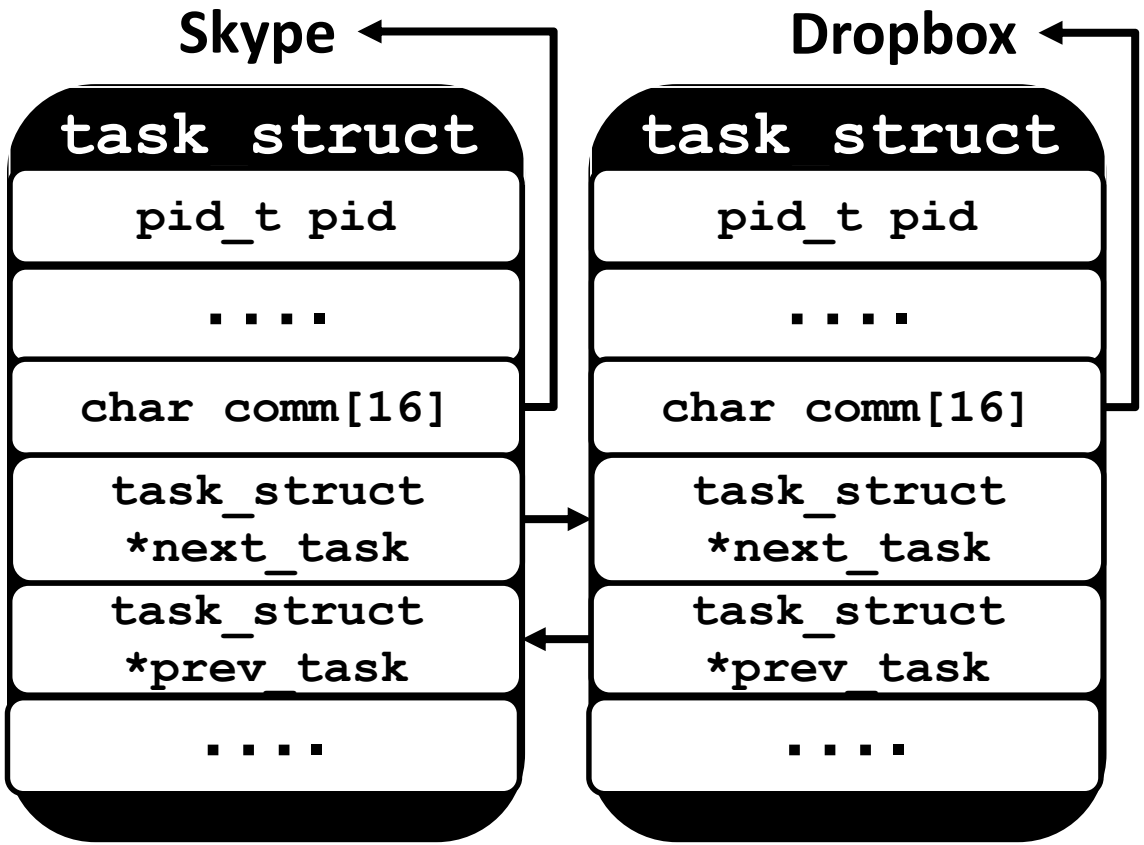


VMI In Action

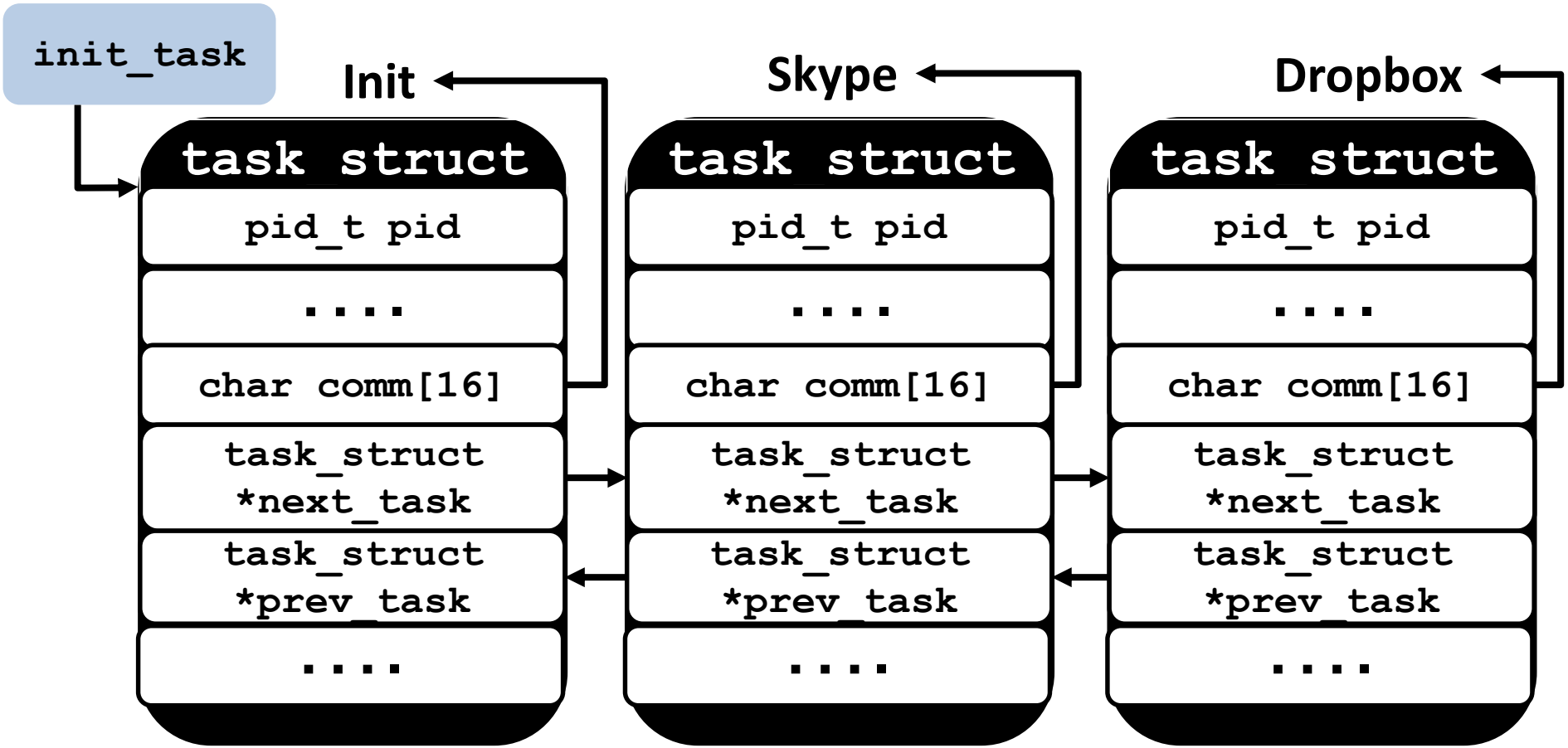


VMI In Action

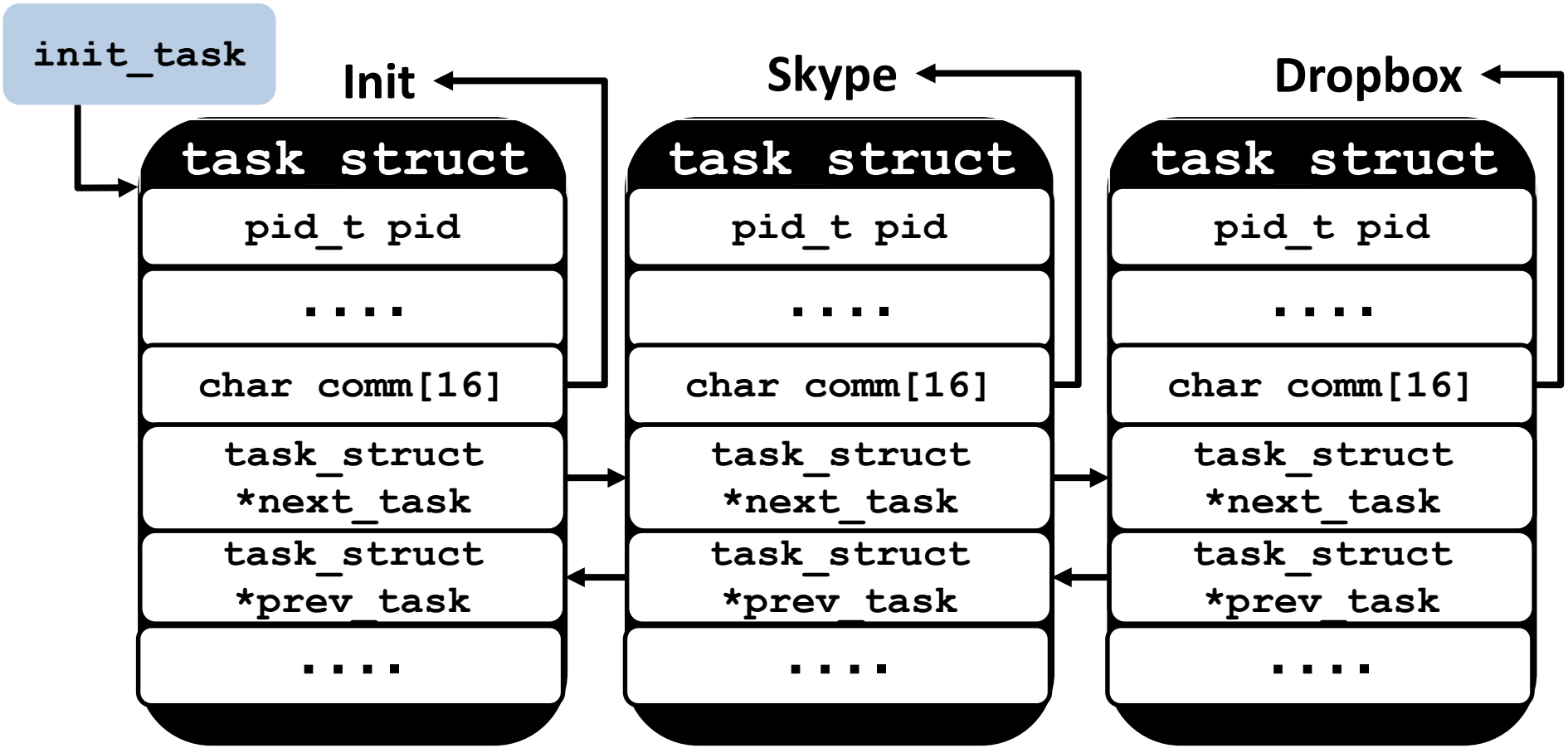
`init_task`



VMI In Action



VMI In Action

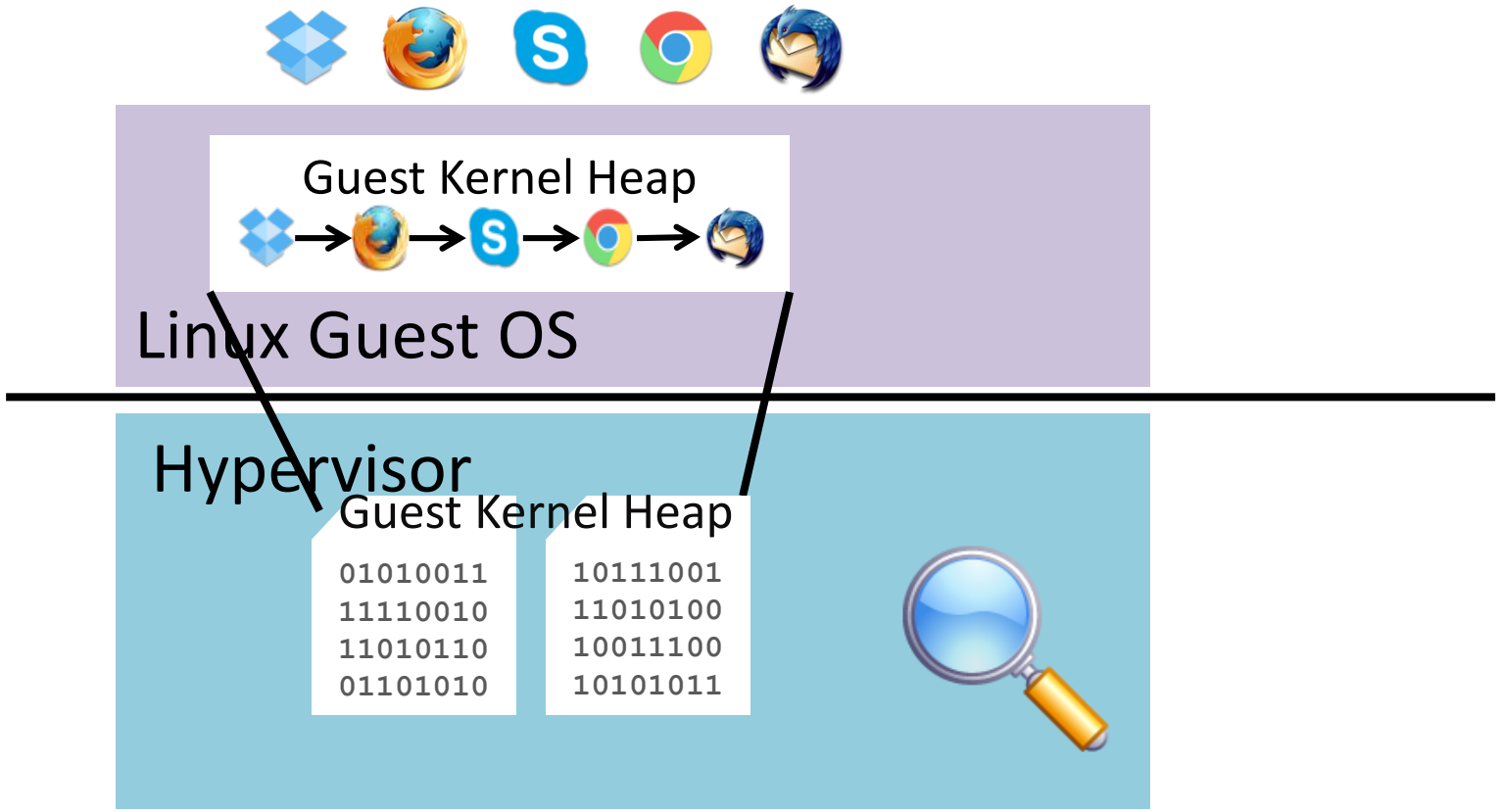


Typecast memory contents to structure definition

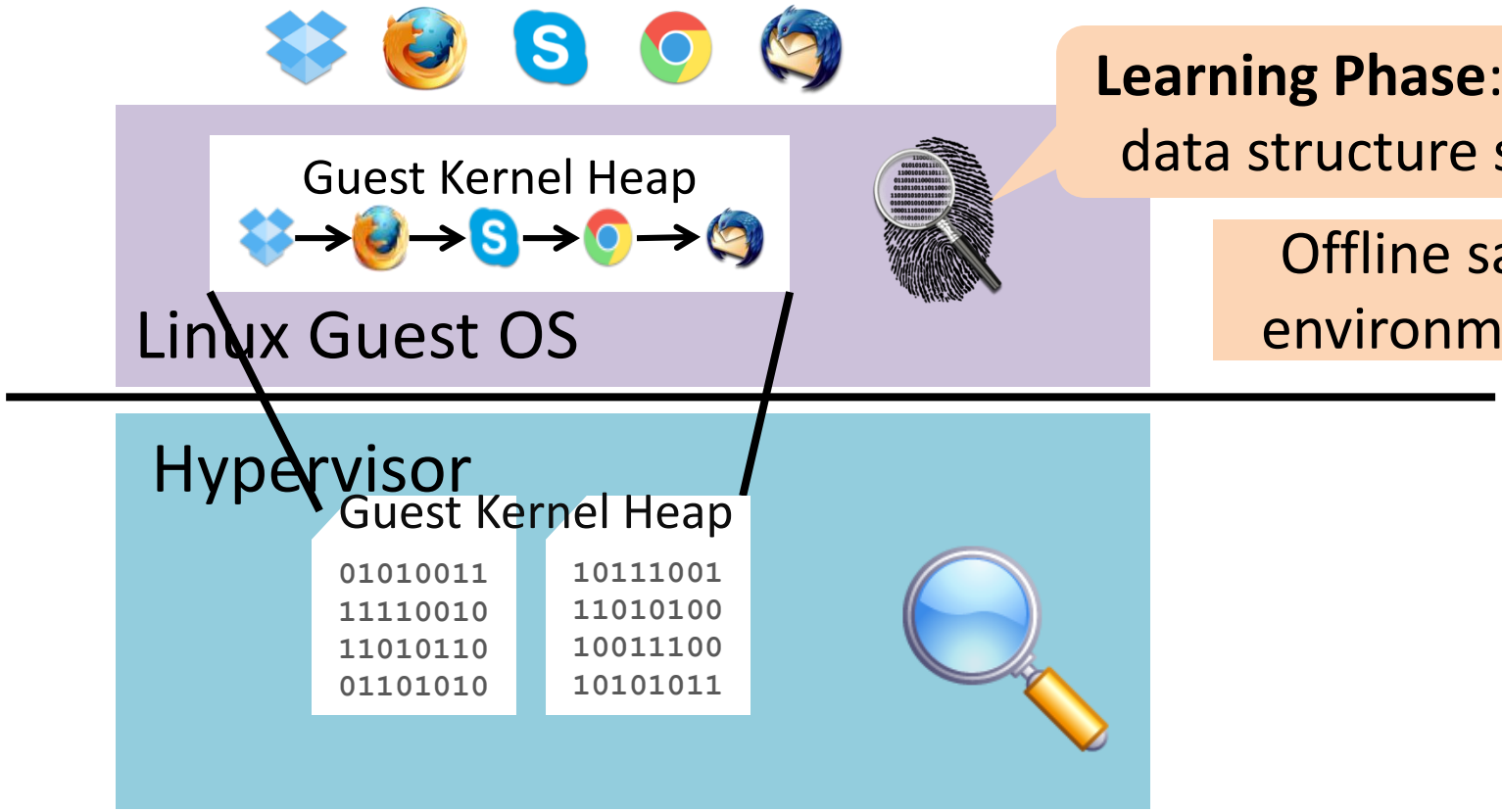
High-level VMI Techniques

- **Learning and Reconstruction**
  - Learn structure signature; Search object instances
- **Code Implanting**
  - Inject code in guest OS; VMM protects injected code
  - State of Art: SIM [1]
- **Data Outgrafting**
  - Reuse static kernel code; Input runtime heap & data
  - State of Art: VMST [2]

Learning and Reconstruction



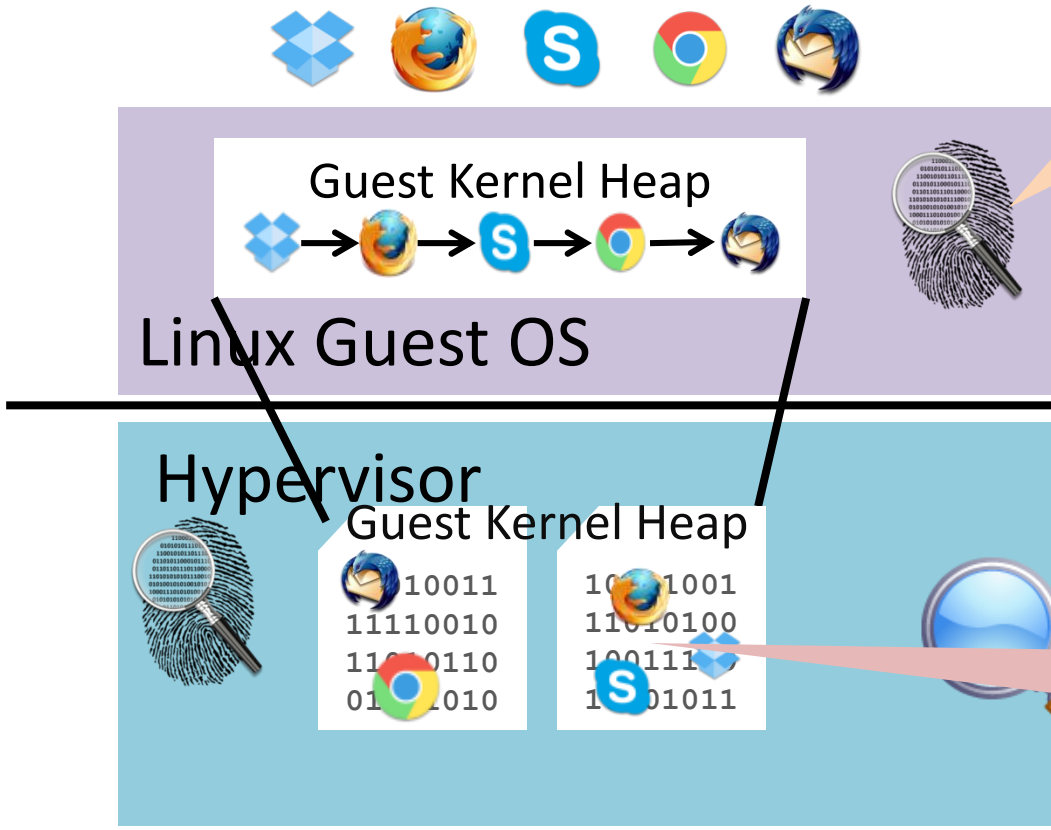
Learning and Reconstruction



Learning Phase: Generate data structure signature

Offline safe environment

Learning and Reconstruction



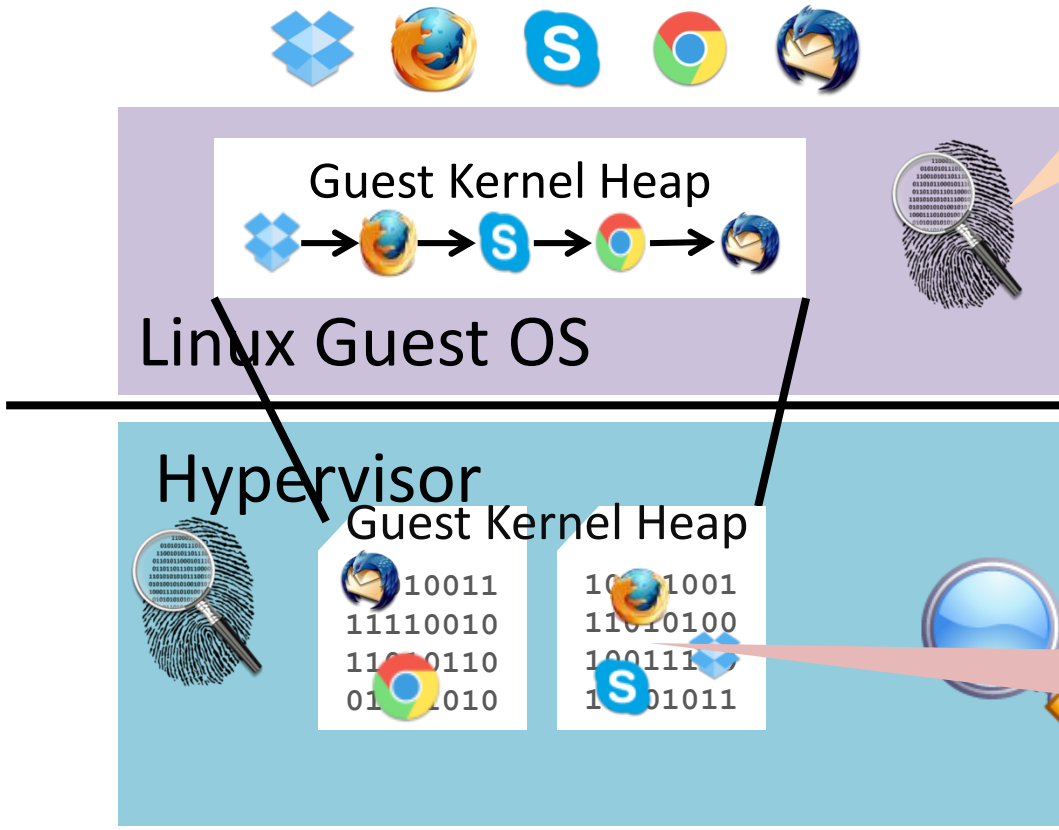
Learning Phase: Generate data structure signature

Offline safe environment

Continuous over guest lifespan

Searching Phase: Find data structure instances

Learning and Reconstruction



Learning Phase: Generate data structure signature

Offline safe environment

Continuous over guest lifespan

Searching Phase: Find data structure instances

Assumption: Same OS behavior in learning and monitoring

Approaches for L&R

- **Learning**: Extract data structure signature
  -
  -
  -
- **Searching**: Identify data structure instances
  -
  -
  -

Approaches for L&R

- **Learning**: Extract data structure signature
  - Hand-crafted data structure signatures
  - .
  - .
- **Searching**: Identify data structure instances
  - .
  - .
  - .

Approaches for L&R

- **Learning**: Extract data structure signature
  - Hand-crafted data structure signatures
  - Source code analysis
  -
- **Searching**: Identify data structure instances
  -
  -
  -

Approaches for L&R

- **Learning**: Extract data structure signature
  - Hand-crafted data structure signatures
  - Source code analysis
  - Dynamic Learning
- **Searching**: Identify data structure instances
  -
  -
  -

Approaches for L&R

- **Learning**: Extract data structure signature
  - Hand-crafted data structure signatures
  - Source code analysis
  - Dynamic Learning
- **Searching**: Identify data structure instances
  - Linearly scan kernel memory
  - .
  - .

Approaches for L&R

- **Learning**: Extract data structure signature
  - Hand-crafted data structure signatures
  - Source code analysis
  - Dynamic Learning
- **Searching**: Identify data structure instances
  - Linearly scan kernel memory
  - Traverse data structure pointers
  -

Approaches for L&R

- **Learning**: Extract data structure signature
  - Hand-crafted data structure signatures
  - Source code analysis
  - Dynamic Learning
- **Searching**: Identify data structure instances
  - Linearly scan kernel memory
  - Traverse data structure pointers
  - Monitor object allocators

Learning Techniques

- Hand-crafted data structure signatures
  -
  -
- Source code analysis
  -
  -
  -
- Dynamic Learning
  -
  -

Learning Techniques

- Hand-crafted data structure signatures
  - Change to an OS kernel requires expert to update tools
  - State of Art: FACE/Ramparser [3], Volatility [4]
- Source code analysis
  -
  -
  -
- Dynamic Learning
  -
  -

Learning Techniques

- Hand-crafted data structure signatures
  - Change to an OS kernel requires expert to update tools
  - State of Art: FACE/Ramparser [3], Volatility [4]
- Source code analysis
  - Points-to analysis generates graph of kernel object types
  - Not all pointers in a data structure point to valid data
  - State of Art: MAS [5], SigGraph [6]
- Dynamic Learning
  -
  -

Learning Techniques

- Hand-crafted data structure signatures
  - Change to an OS kernel requires expert to update tools
  - State of Art: FACE/Ramparser [3], Volatility [4]
- Source code analysis
  - Points-to analysis generates graph of kernel object types
  - Not all pointers in a data structure point to valid data
  - State of Art: MAS [5], SigGraph [6]
- Dynamic Learning
  - Supervised machine learning: train on a trusted OS
  - State of Art: RSFKDS [7]

Searching Techniques

- Linearly scan kernel memory

Guest Kernel Heap

```
01010011    10111001
11110010    11010100
11010110    10011100
01101010    10101011
```

- Traverse data structure pointers

Searching Techniques

- Linearly scan kernel memory

Guest Kernel Heap

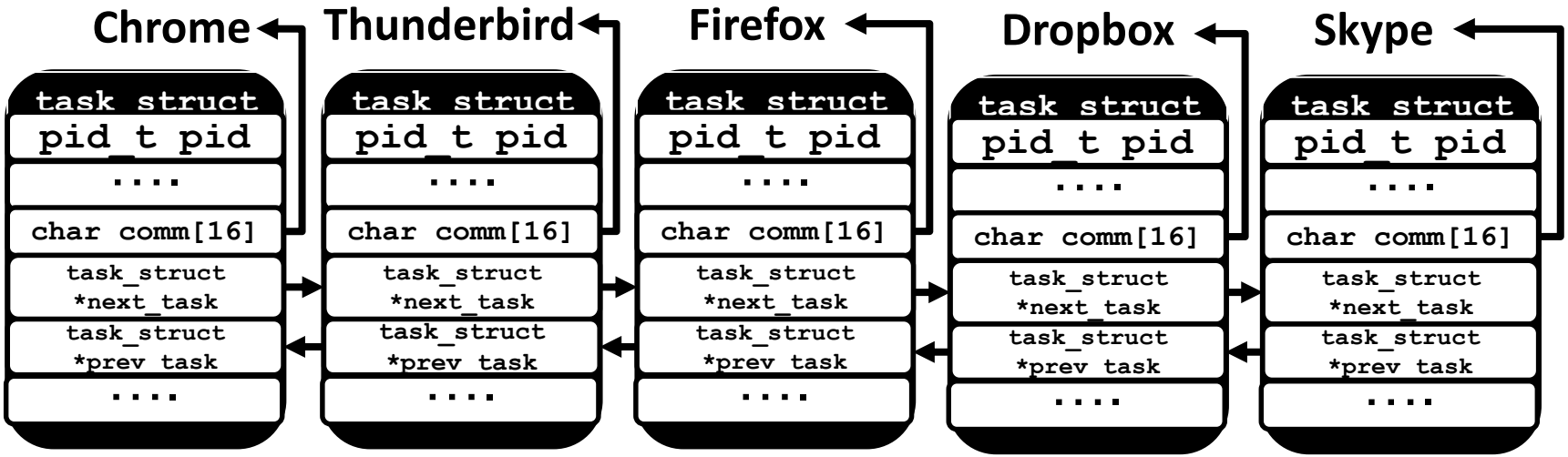
- Traverse data structure pointers

Searching Techniques

- Linearly scan kernel memory

Guest Kernel Heap

- Traverse data structure pointers



But L&R is too involved

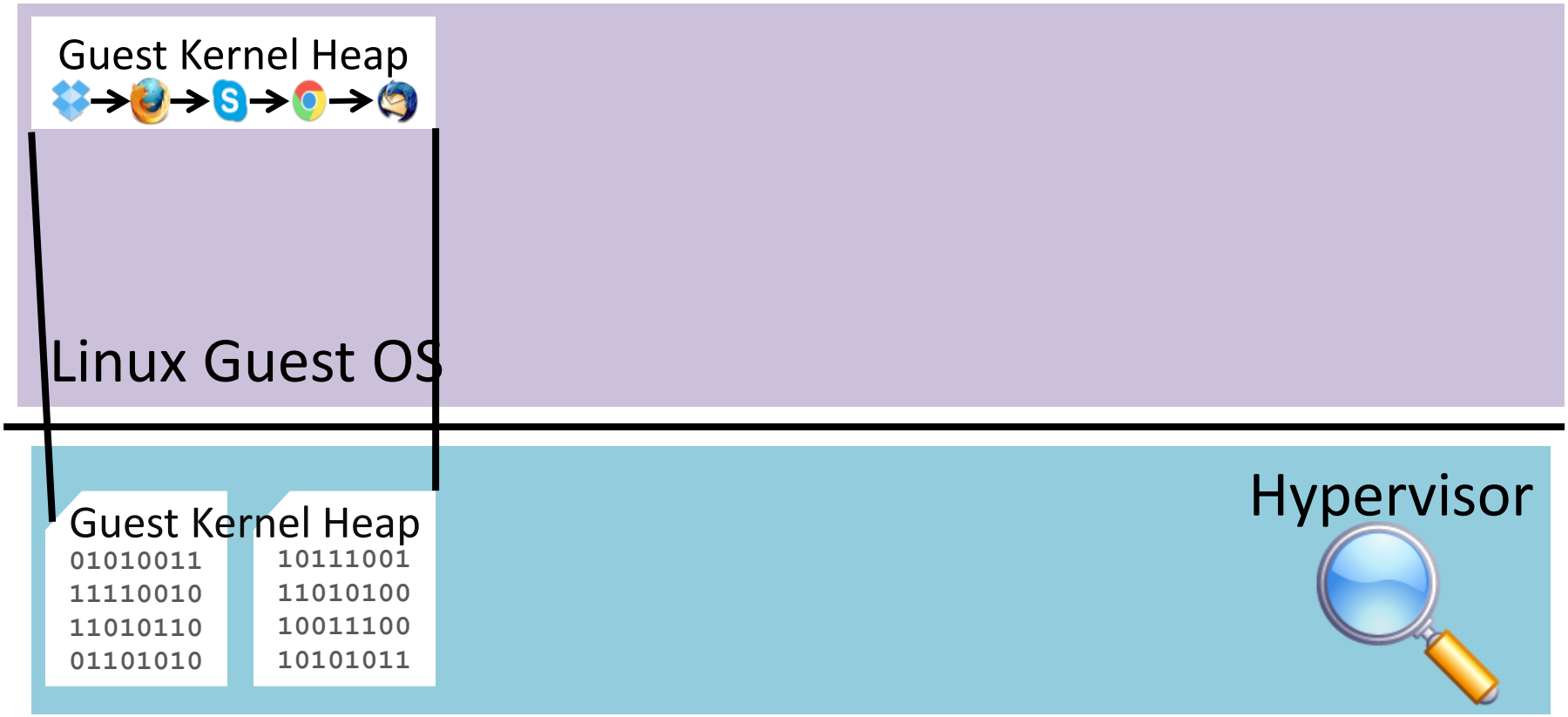
- L&R builds tools to mine information
  - This is hard!!!
- Can we just cheat?
  - Reuse the static guest kernel code
  - Make runtime kernel data and heap available to it

Just reuse the guest code to interpret kernel heap

High-level VMI Techniques

- **Learning and Reconstruction**
  - Learn structure signature; Search object instances
- **Code Implanting**
  - Inject code in guest OS; VMM protects injected code
  - State of Art: SIM [1]
- **Data Outgrafting**
  - Reuse static kernel code; Input runtime heap & data
  - State of Art: VMST [2]

Code Implanting



Code Implanting

Guest Kernel Heap

```
struct task_struct *task;
for_each_process(task)
    call_home("%s\n",task->comm);
return 0;
```

Linux Guest OS

Guest Kernel Heap

```
01010011  10111001
11110010  11010100
11010110  10011100
01101010  10101011
```

Hypervisor

Code Implanting

Guest Kernel Heap

```
struct task_struct *task;
for_each_process(task)
    call_home("%s\n",task->comm);
return 0;
```

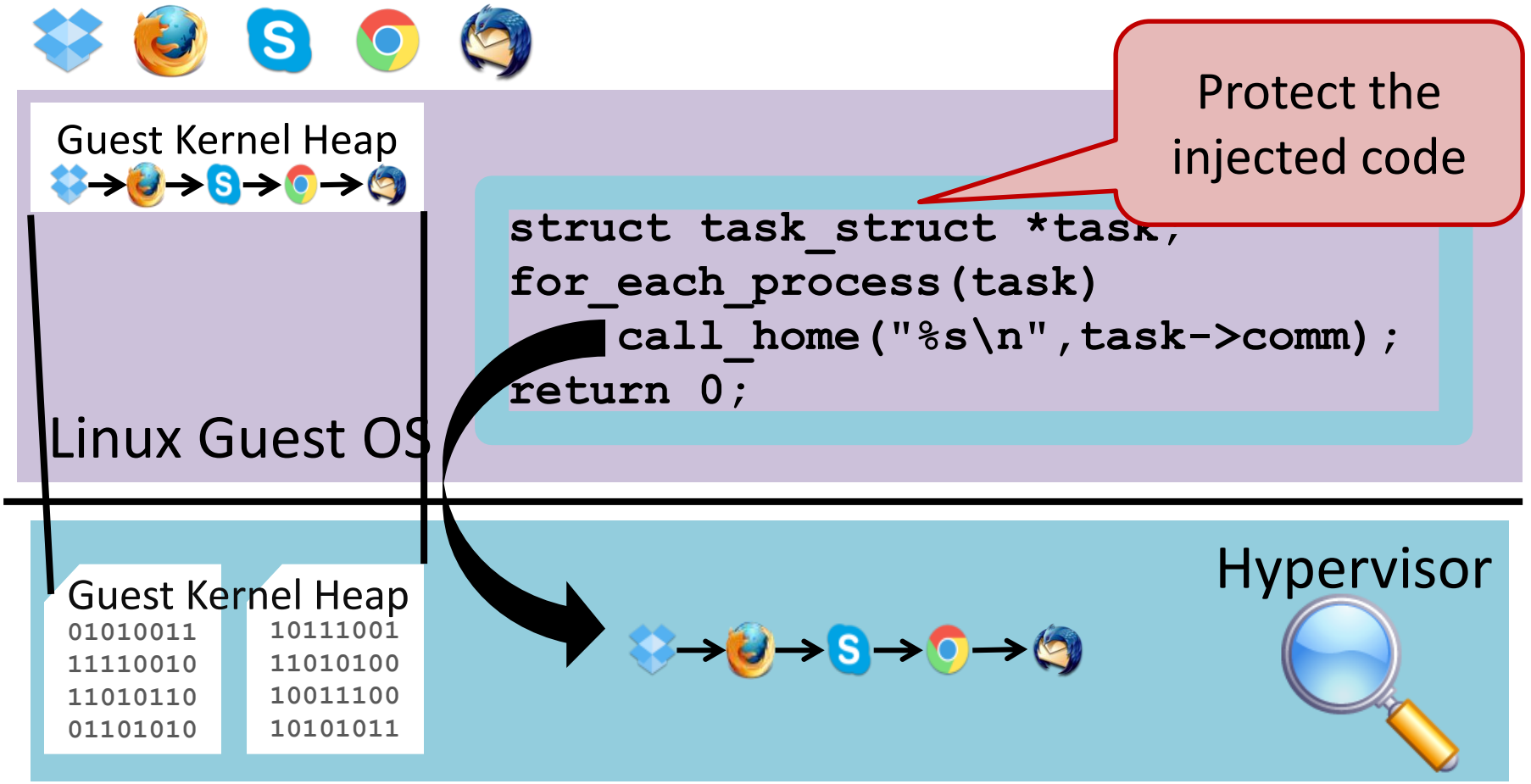
Linux Guest OS

Guest Kernel Heap

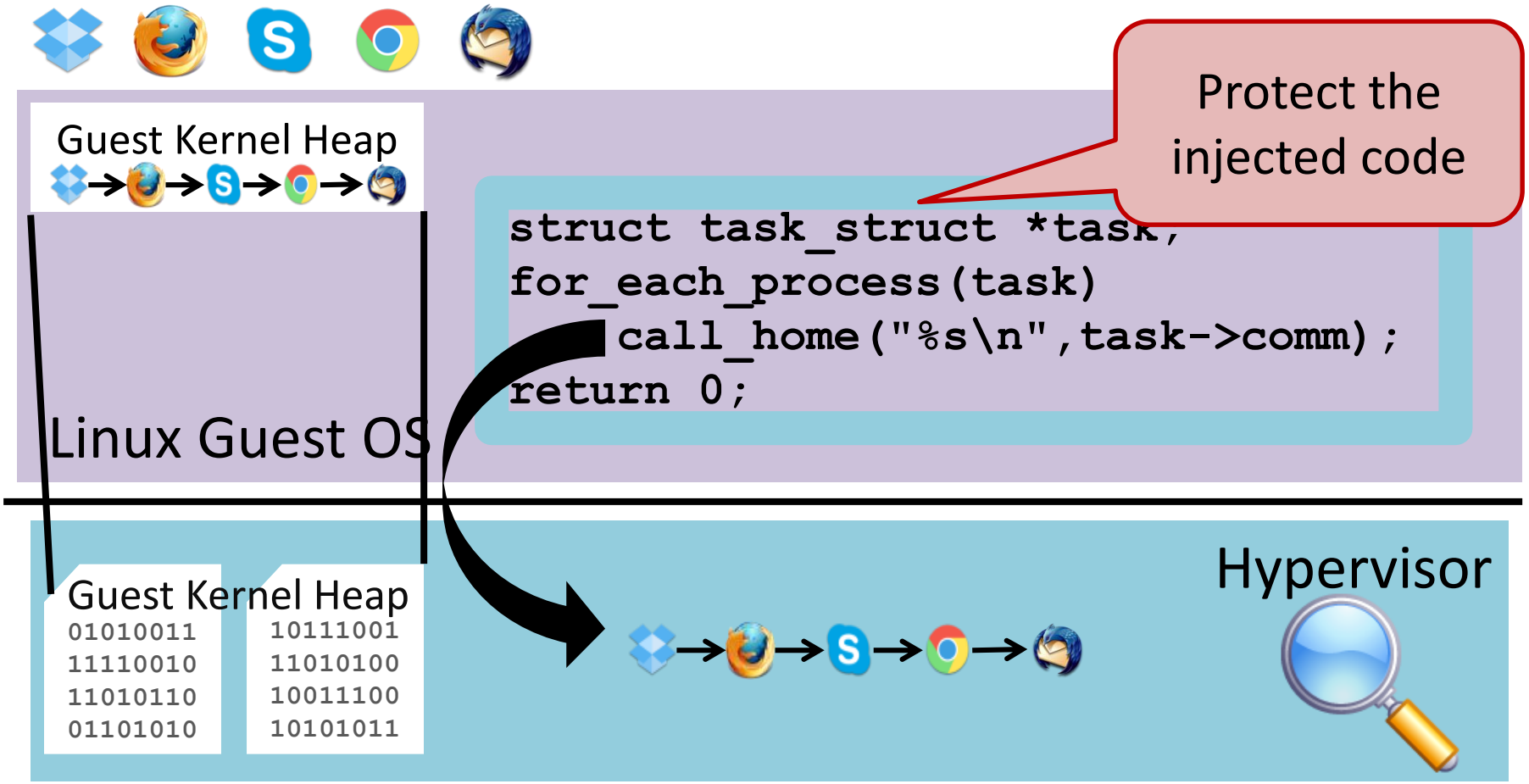
01010011	10111001
11110010	11010100
11010110	10011100
01101010	10101011

Hypervisor

Code Implanting



Code Implanting

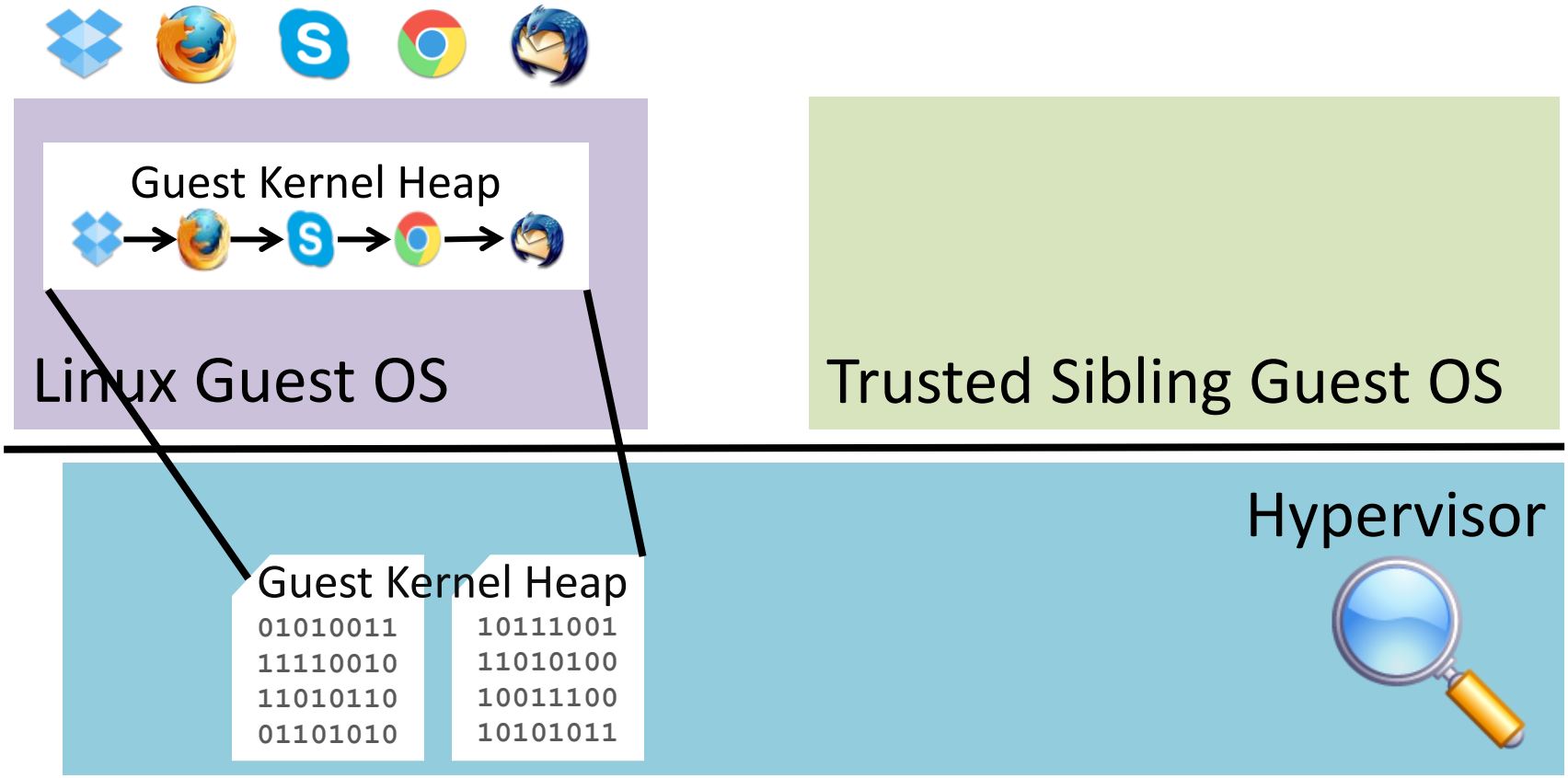


Inject code in guest OS; Difficult to protect

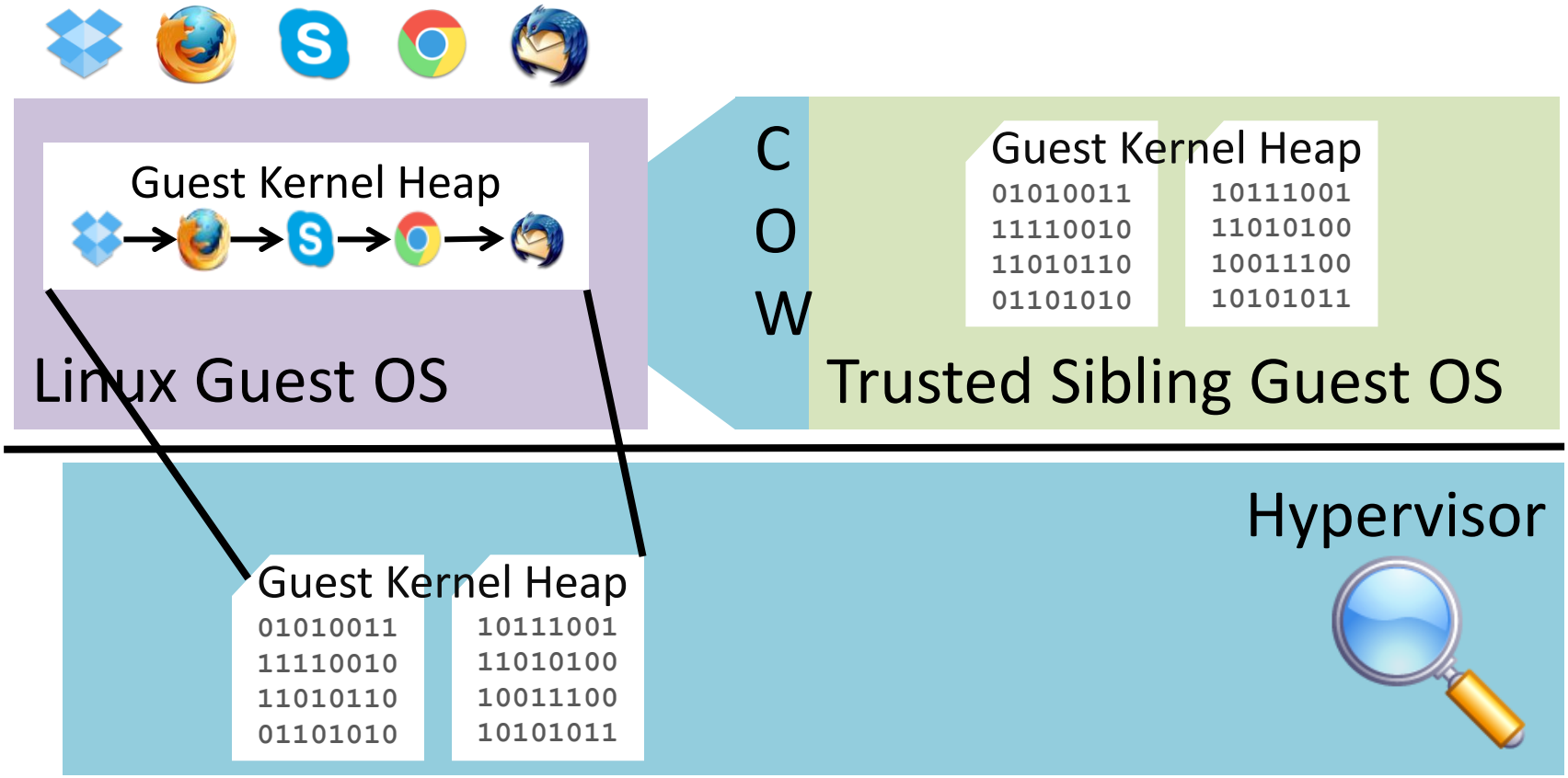
High-level VMI Techniques

- **Learning and Reconstruction**
  - Learn structure signature; Search object instances
- **Code Implanting**
  - Inject code in guest OS; VMM protects injected code
  - State of Art: SIM [1]
- **Data Outgrafting**
  - Reuse static kernel code; Input runtime heap & data
  - State of Art: VMST [2]

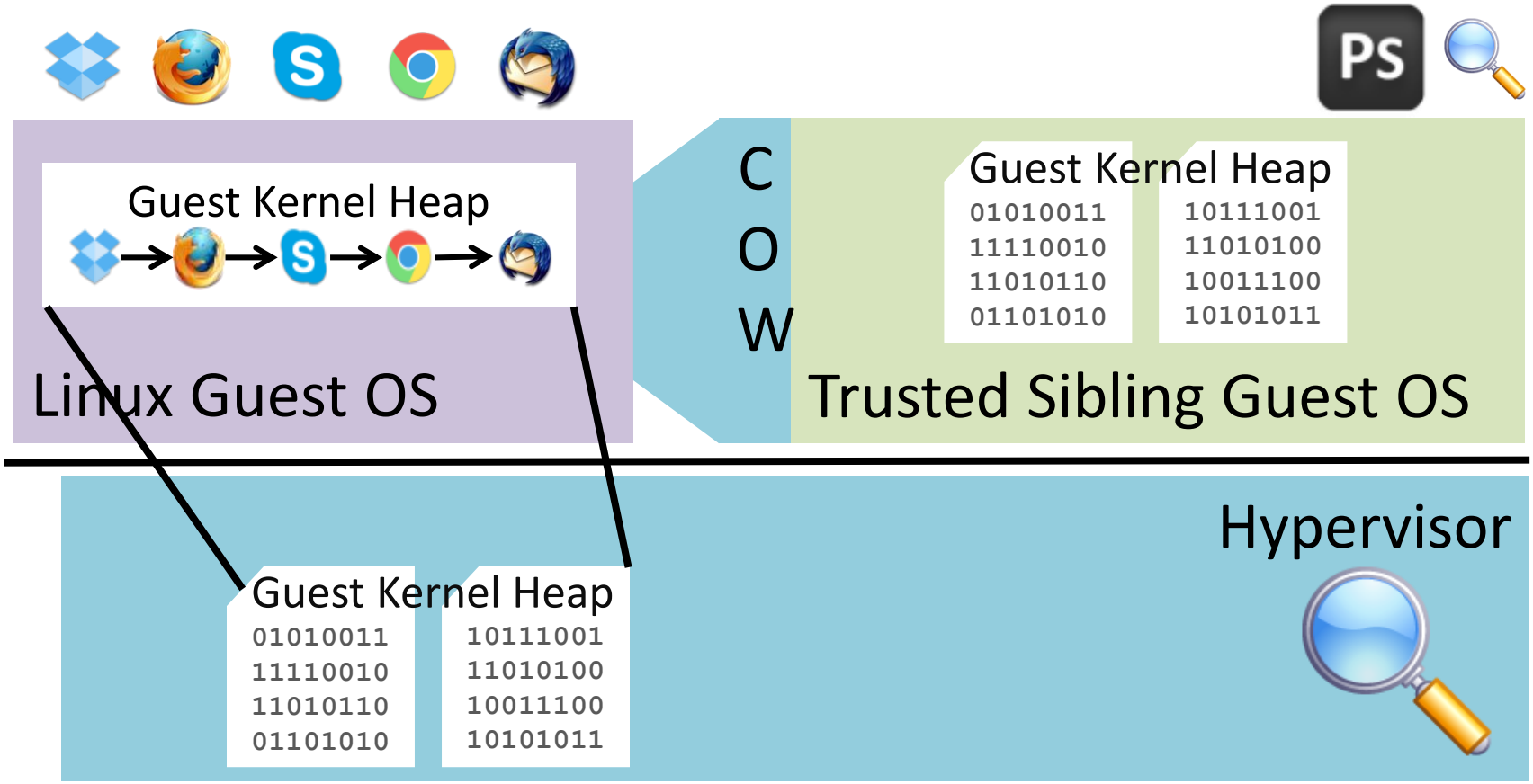
Data Outgrafting



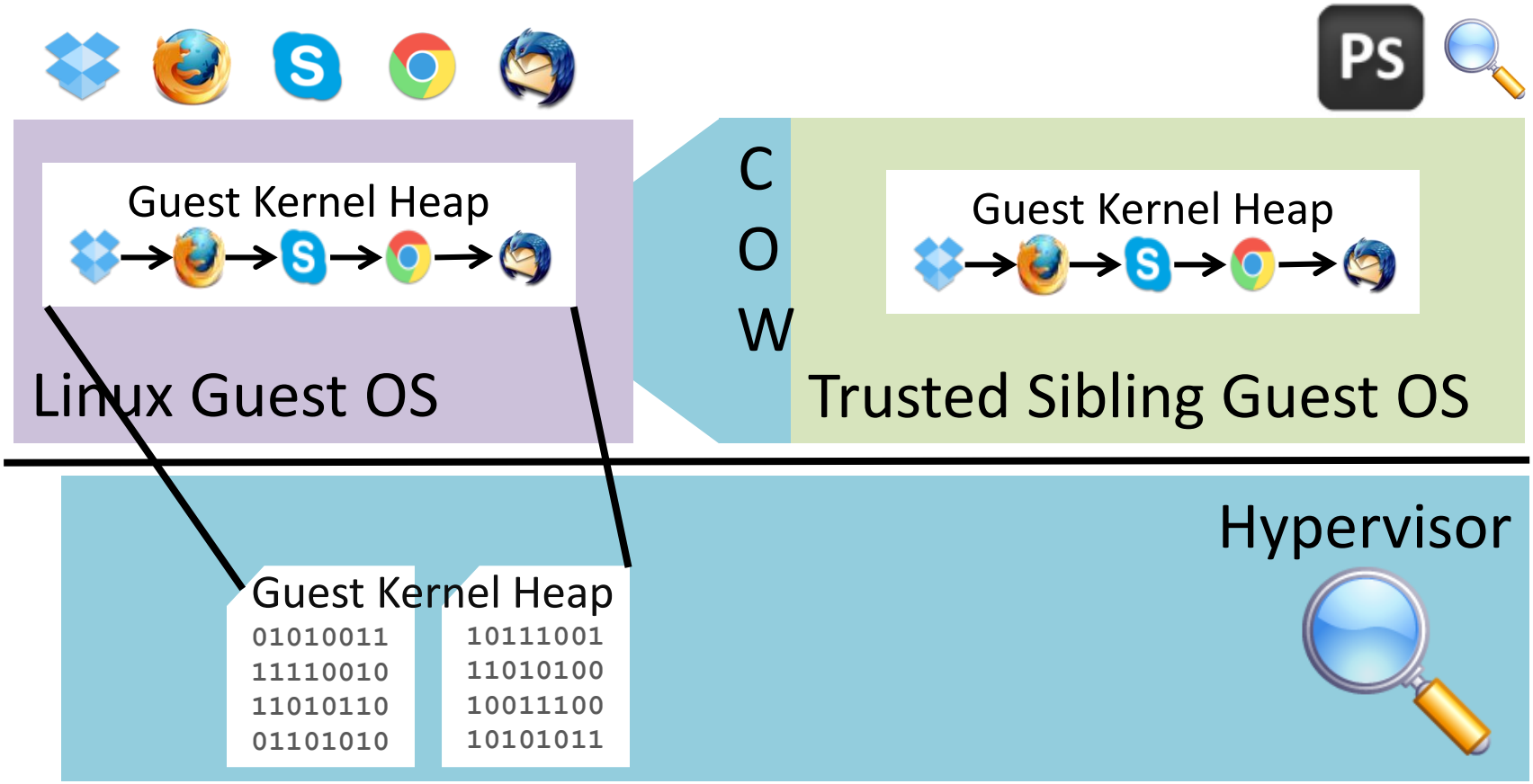
Data Outgrafting



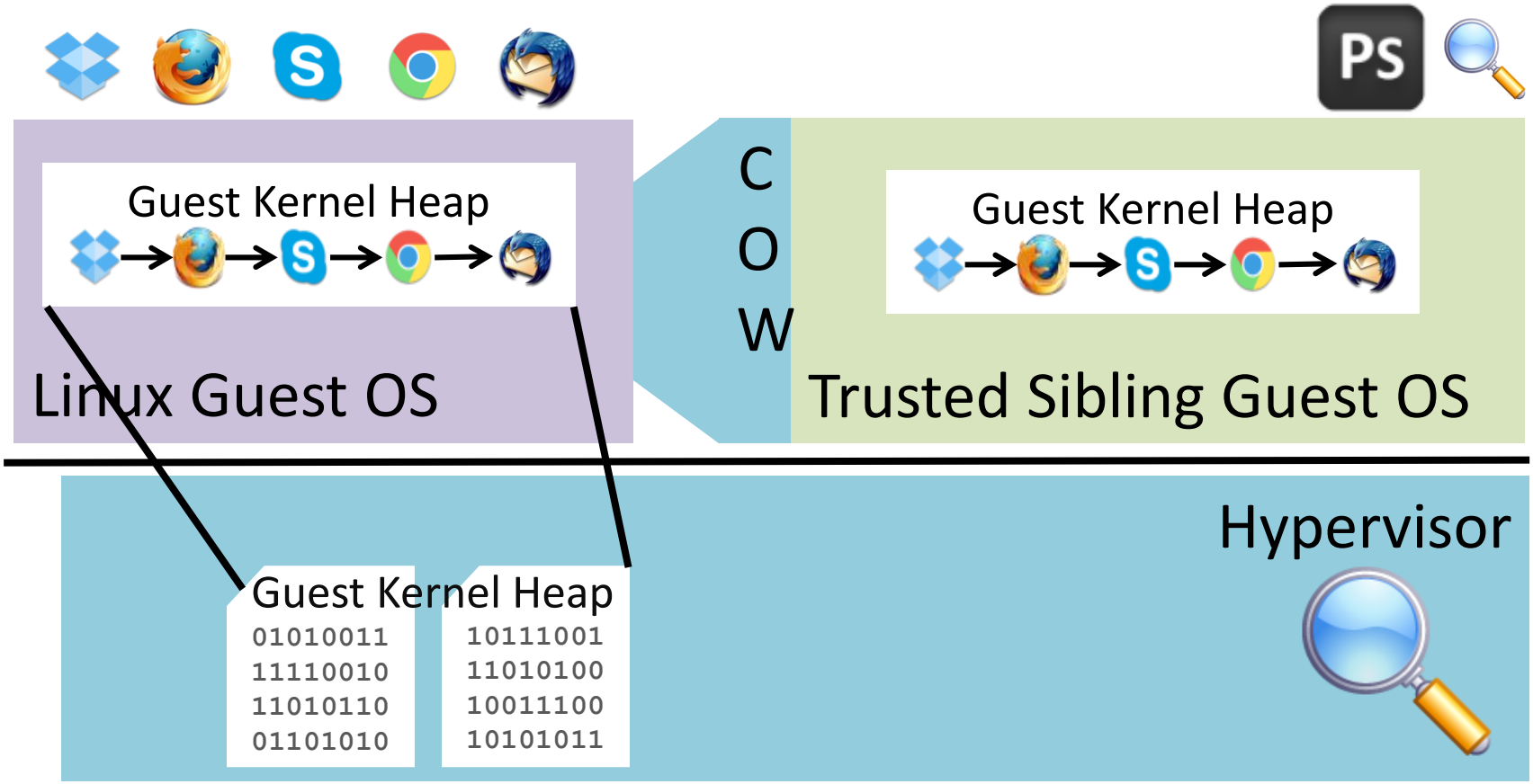
Data Outgrafting



Data Outgrafting

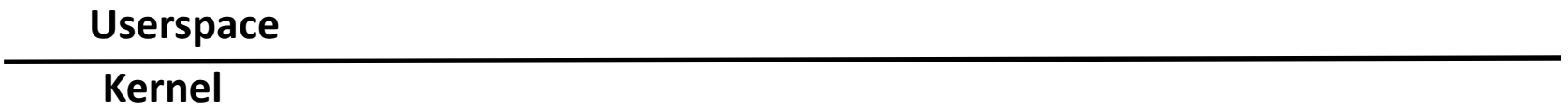


Data Outgrafting

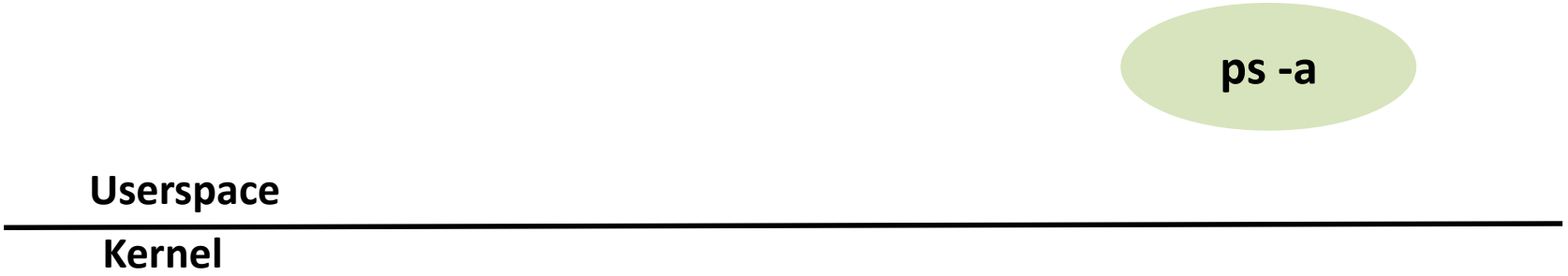


Reuse static trusted kernel code; Input runtime heap & data

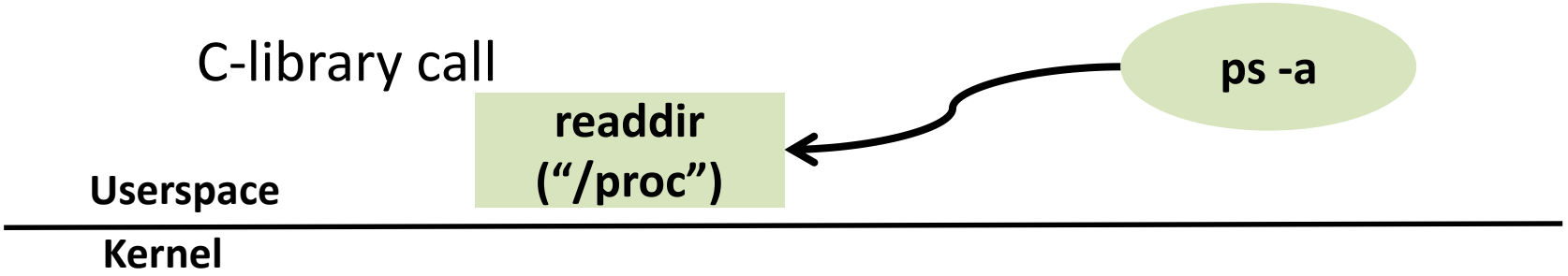
Listing all the running processes



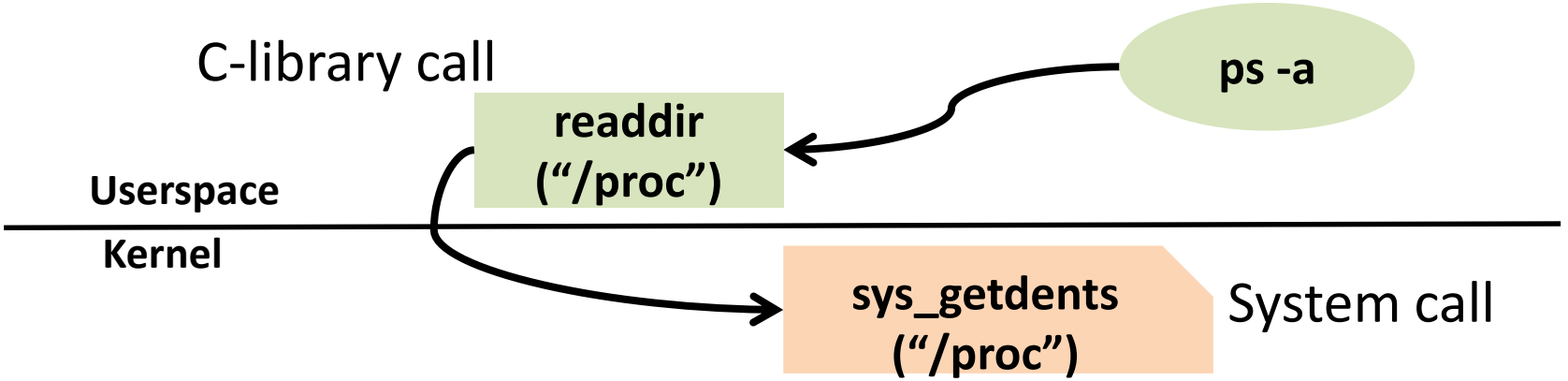
Listing all the running processes



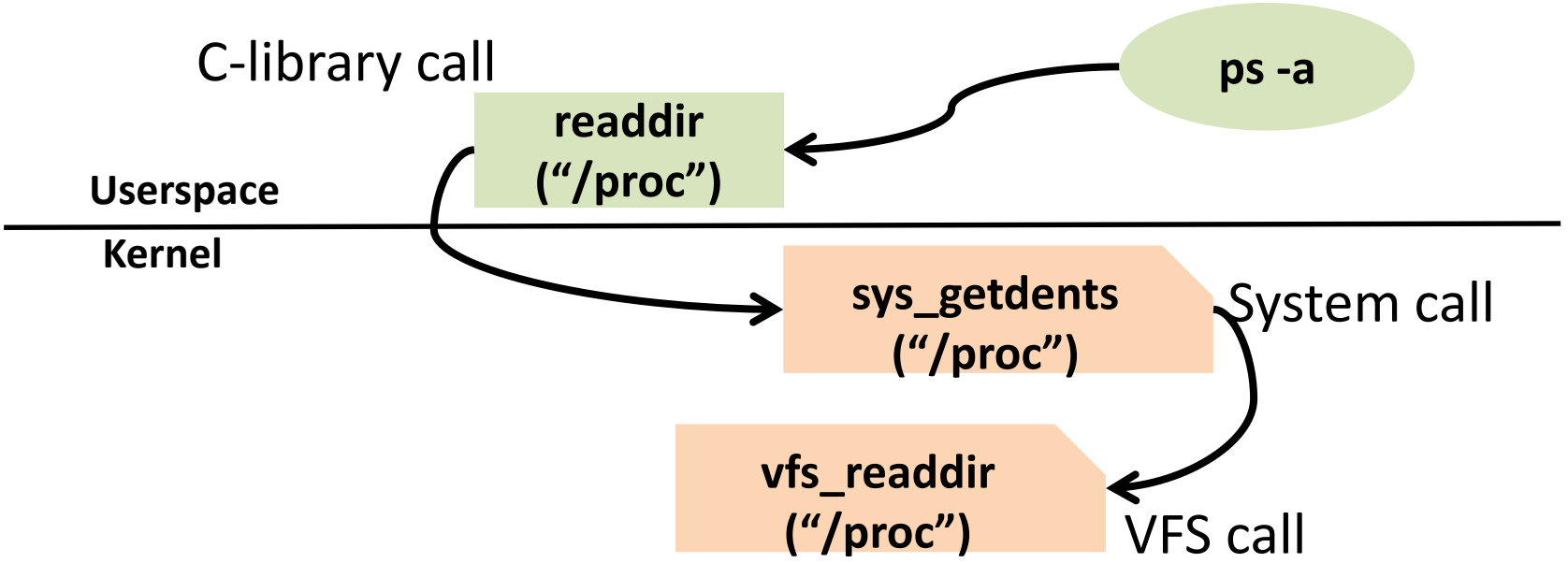
Listing all the running processes



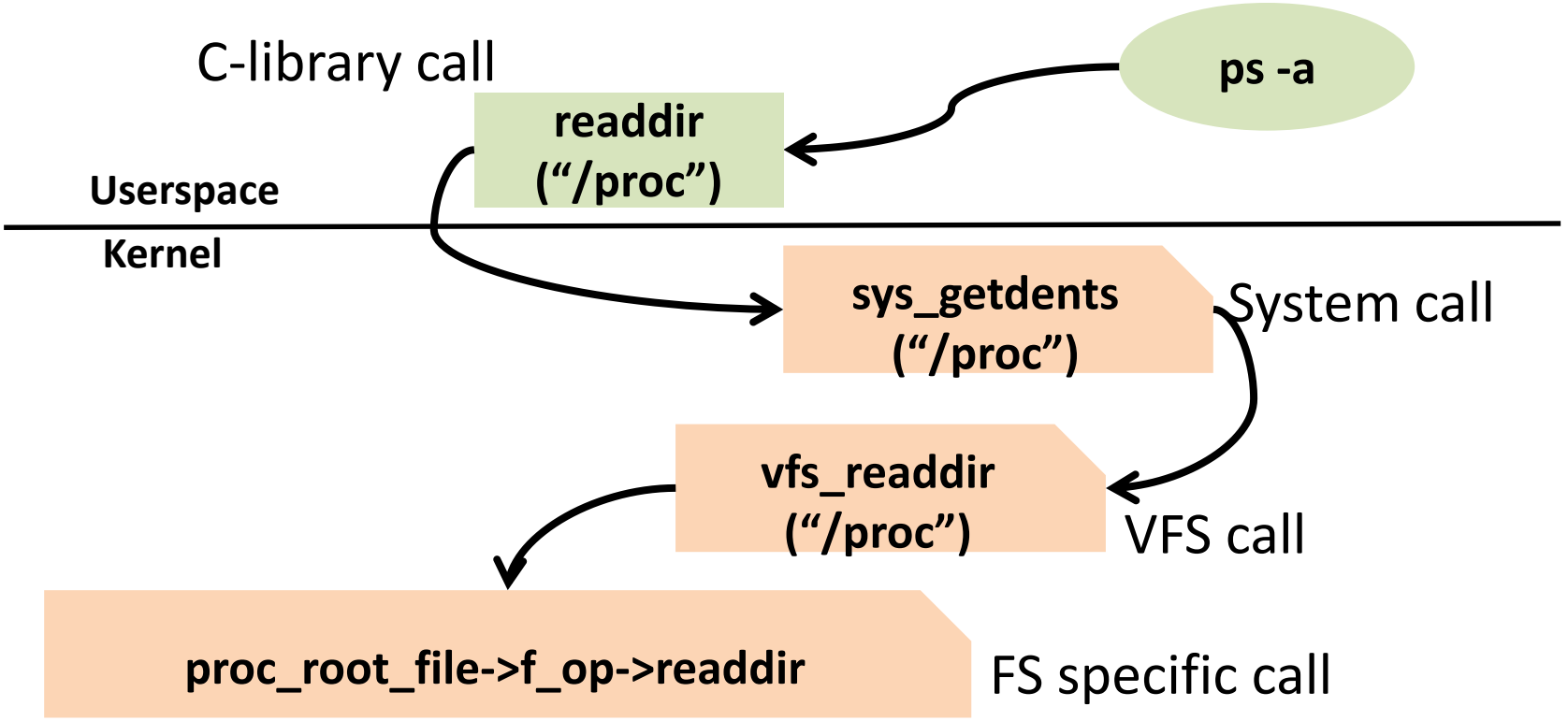
Listing all the running processes



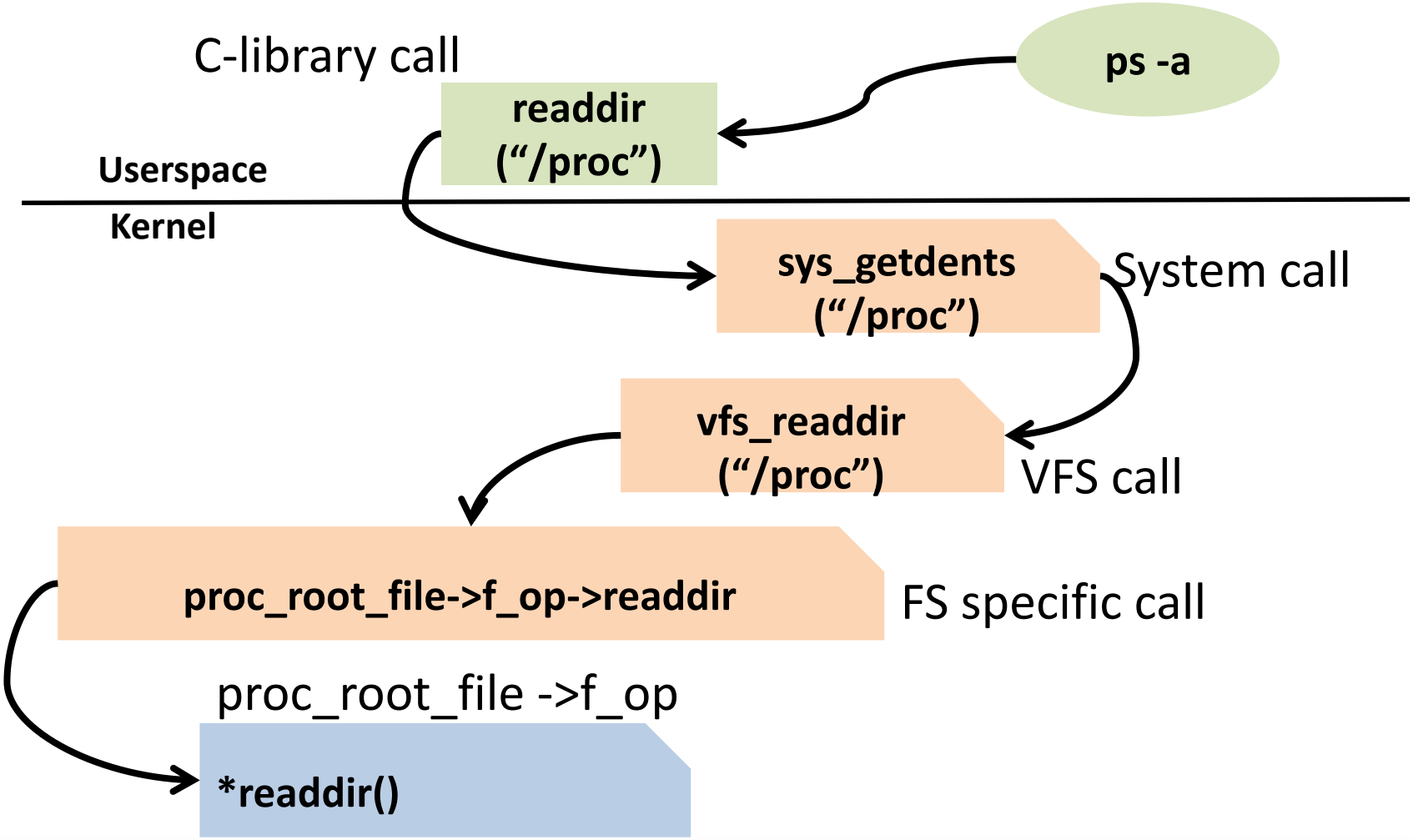
Listing all the running processes



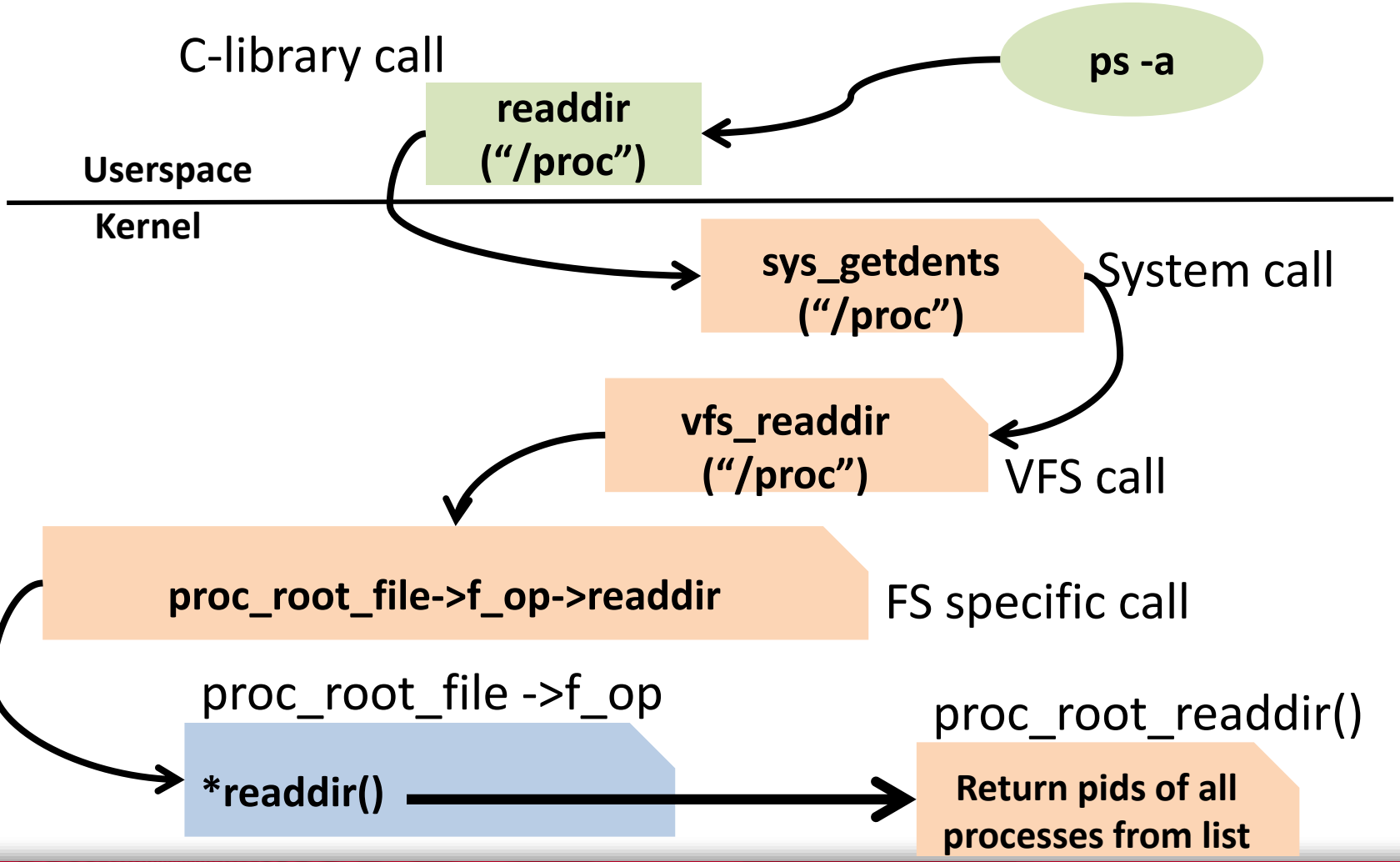
Listing all the running processes



Listing all the running processes



Listing all the running processes

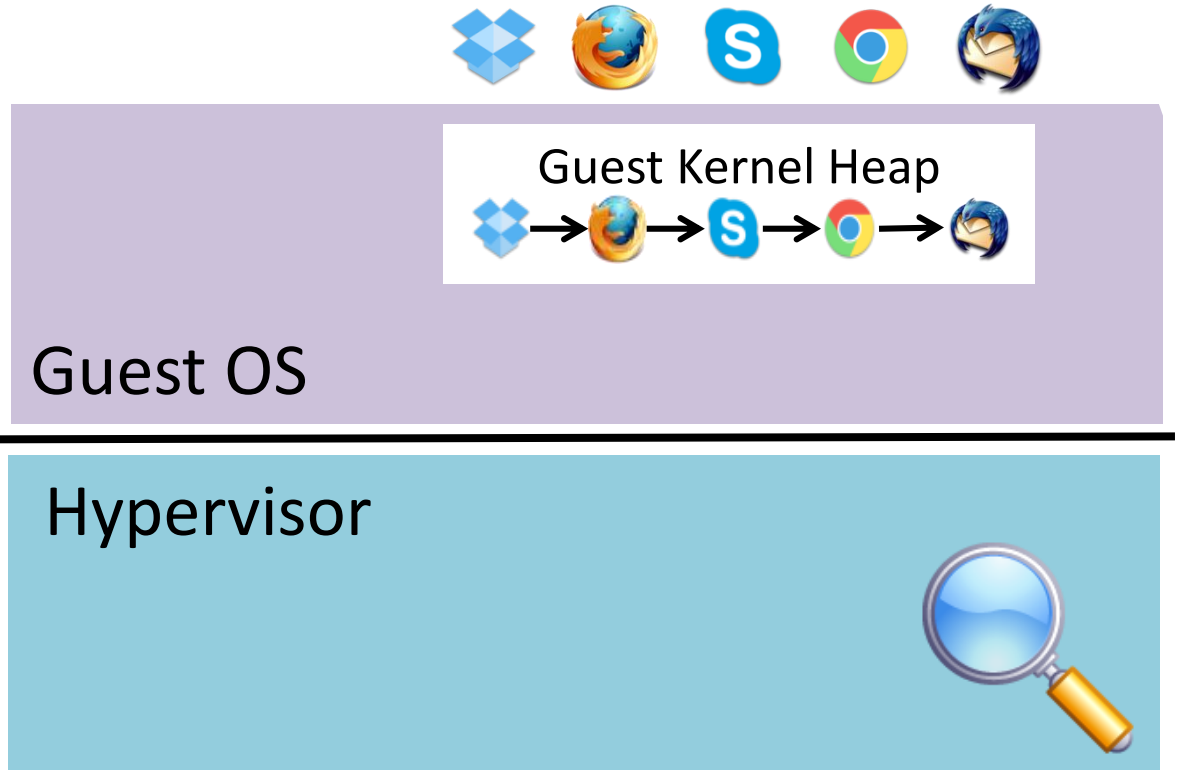


Trusted Guest Pervasive in VMI

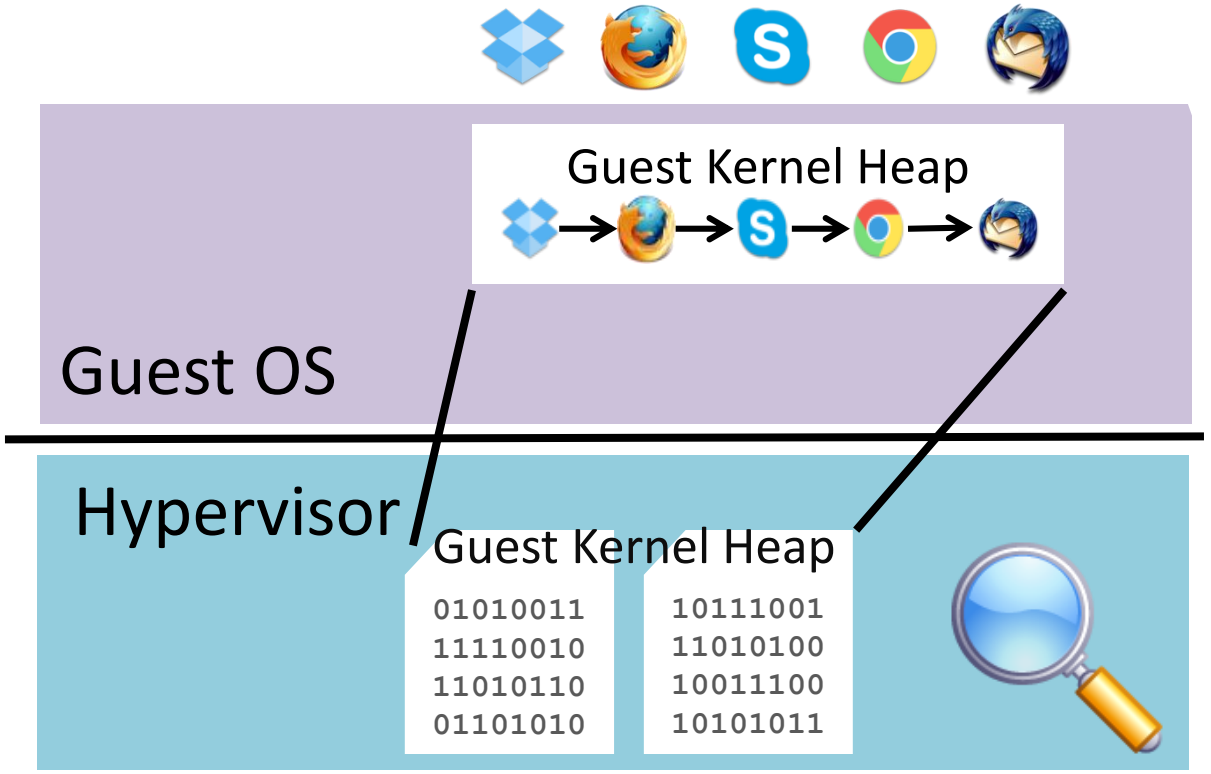
Technique	Approach	Trust Guest That
Learn & Reconstruct	Learn template then search	➢Same OS behavior in learning and monitoring phases
Code Implanting	Monitoring inside guest OS	➢Guest OS reports correct information
Data Outgrafting	Use sibling VM; share memory	➢Identical guest OS behavior in monitored and trusted sibling VM

Current VMI techniques built on some level of trust in guest

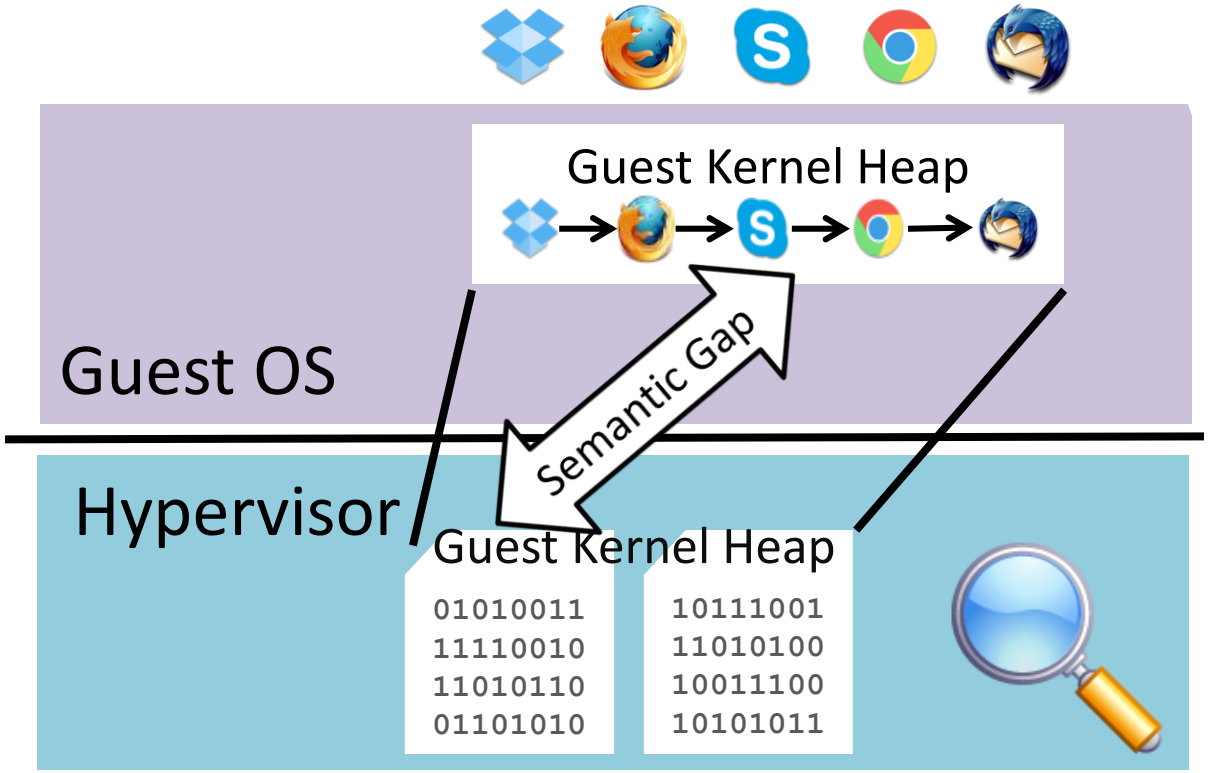
Semantic Gap



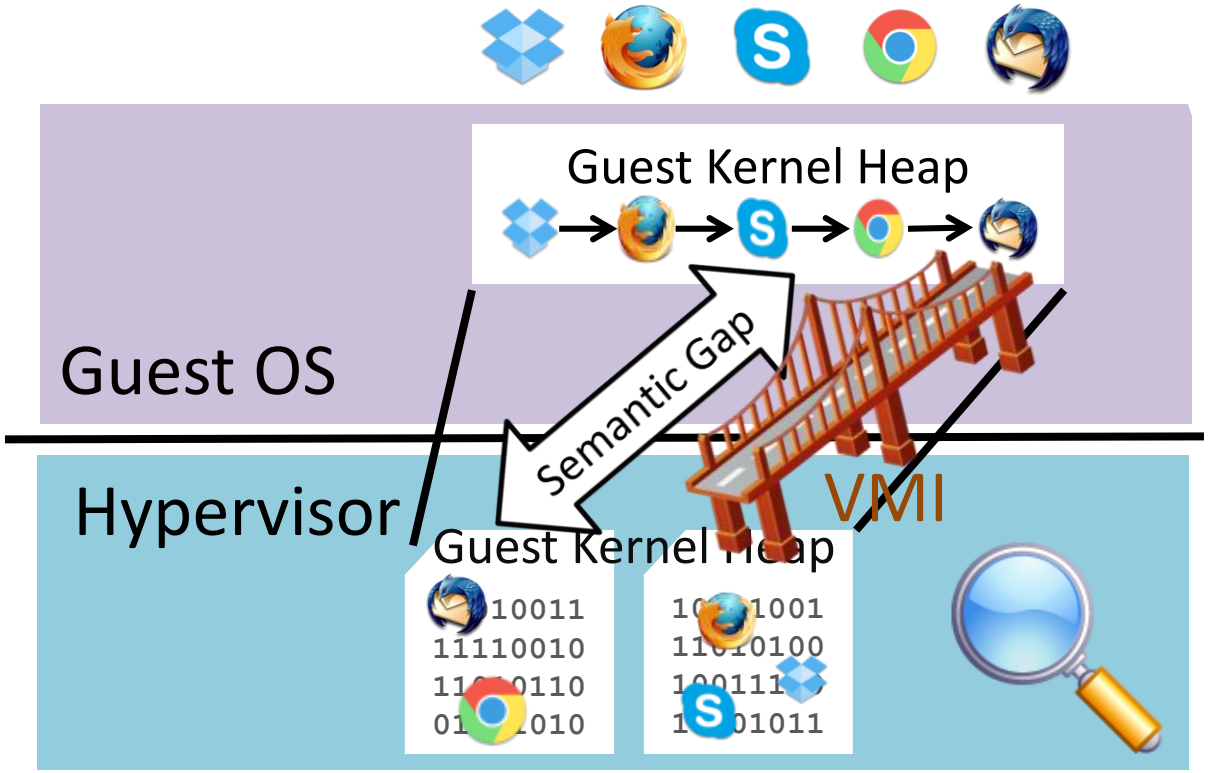
Semantic Gap



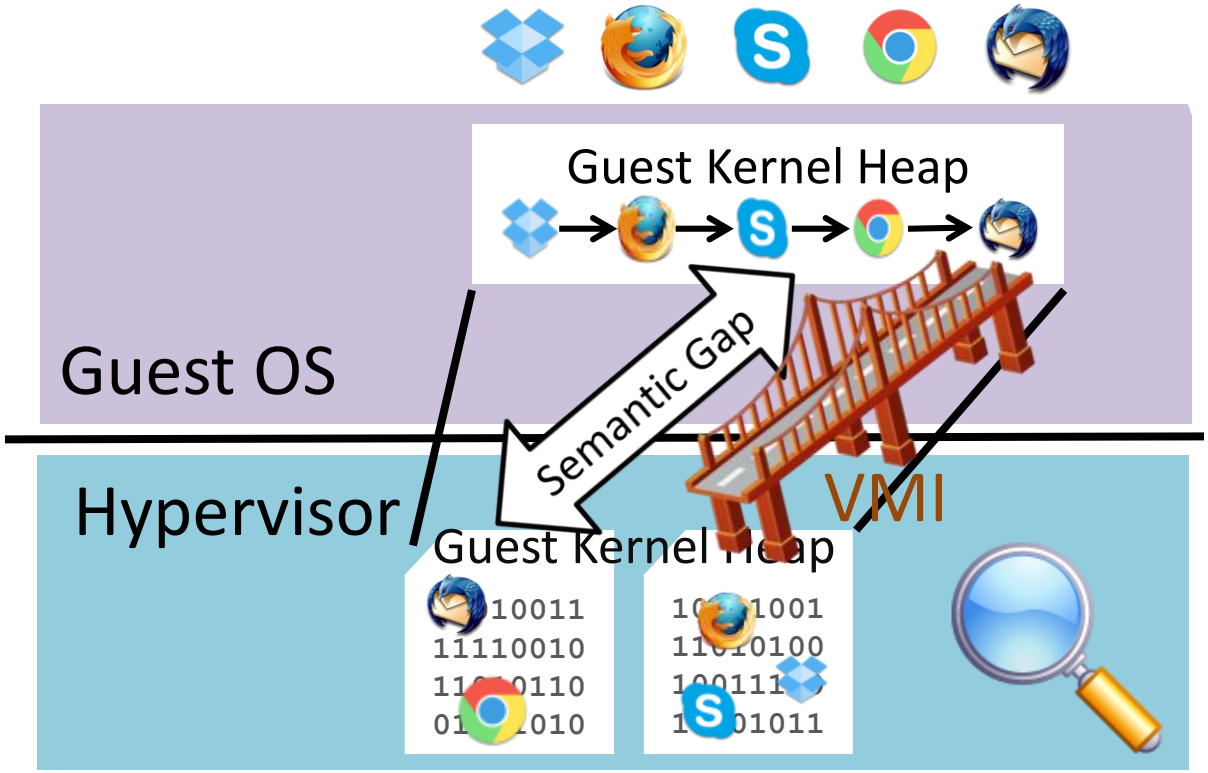
Semantic Gap



Semantic Gap



Semantic Gap



Need high level information; Available low level information

Semantic Gap: Details

Semantic Gap: Details

Hypervisor can observe	Monitor wants to observe
	Variable values

Semantic Gap: Details

Hypervisor can observe	Monitor wants to observe
CPU Registers	Variable values

Semantic Gap: Details

Hypervisor can observe	Monitor wants to observe
CPU Registers	Variable values
	Objects and Types

Semantic Gap: Details

Hypervisor can observe	Monitor wants to observe
CPU Registers	Variable values
Physical Memory	Objects and Types

Semantic Gap: Details

Hypervisor can observe	Monitor wants to observe
CPU Registers	Variable values
Physical Memory	Objects and Types
	File system and Files

Semantic Gap: Details

Hypervisor can observe	Monitor wants to observe
CPU Registers	Variable values
Physical Memory	Objects and Types
Disk Data	File system and Files

Semantic Gap: Details

Hypervisor can observe	Monitor wants to observe
CPU Registers	Variable values
Physical Memory	Objects and Types
Disk Data	File system and Files
	Interrupt/Exceptions

Semantic Gap: Details

Hypervisor can observe	Monitor wants to observe
CPU Registers	Variable values
Physical Memory	Objects and Types
Disk Data	File system and Files
Hardware Events	Interrupt/Exceptions

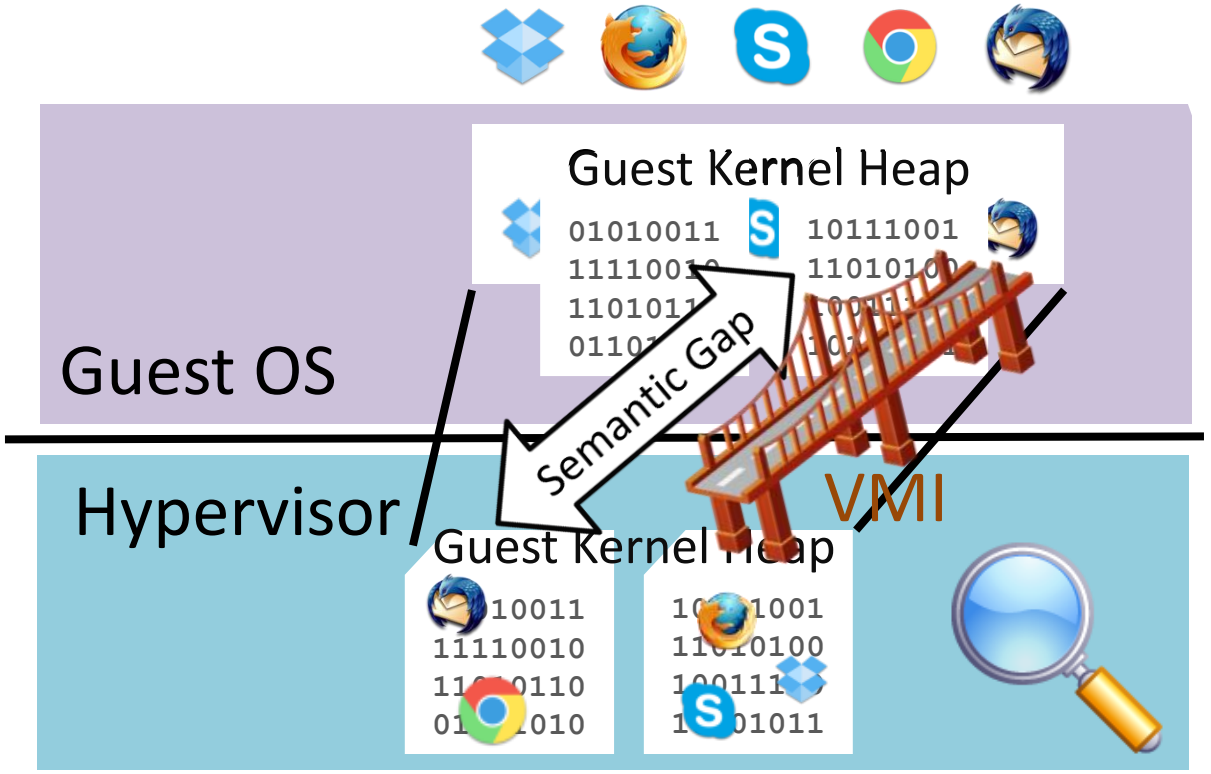
Semantic Gap: Details

Hypervisor can observe	Monitor wants to observe
CPU Registers	Variable values
Physical Memory	Objects and Types
Disk Data	File system and Files
Hardware Events	Interrupt/Exceptions
	Packets, Buffers

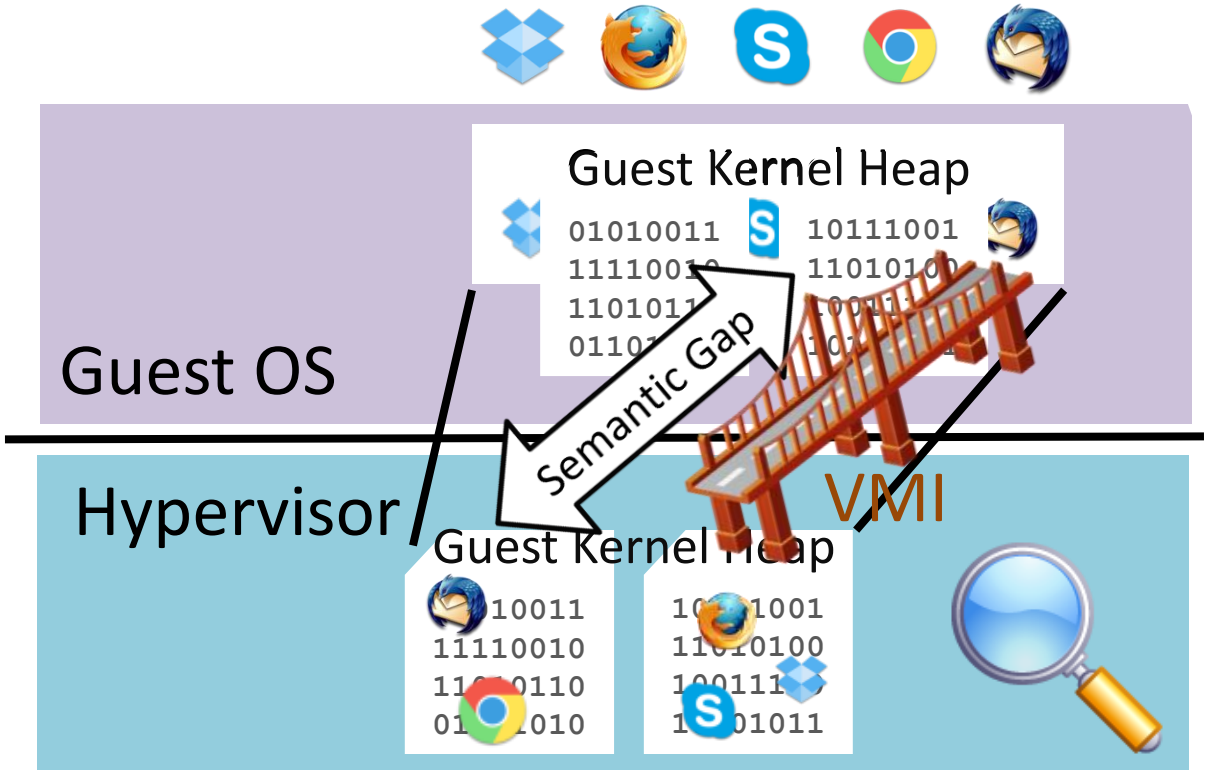
Semantic Gap: Details

Hypervisor can observe	Monitor wants to observe
CPU Registers	Variable values
Physical Memory	Objects and Types
Disk Data	File system and Files
Hardware Events	Interrupt/Exceptions
I/O Data	Packets, Buffers

Semantic Gap: A challenge for VMI

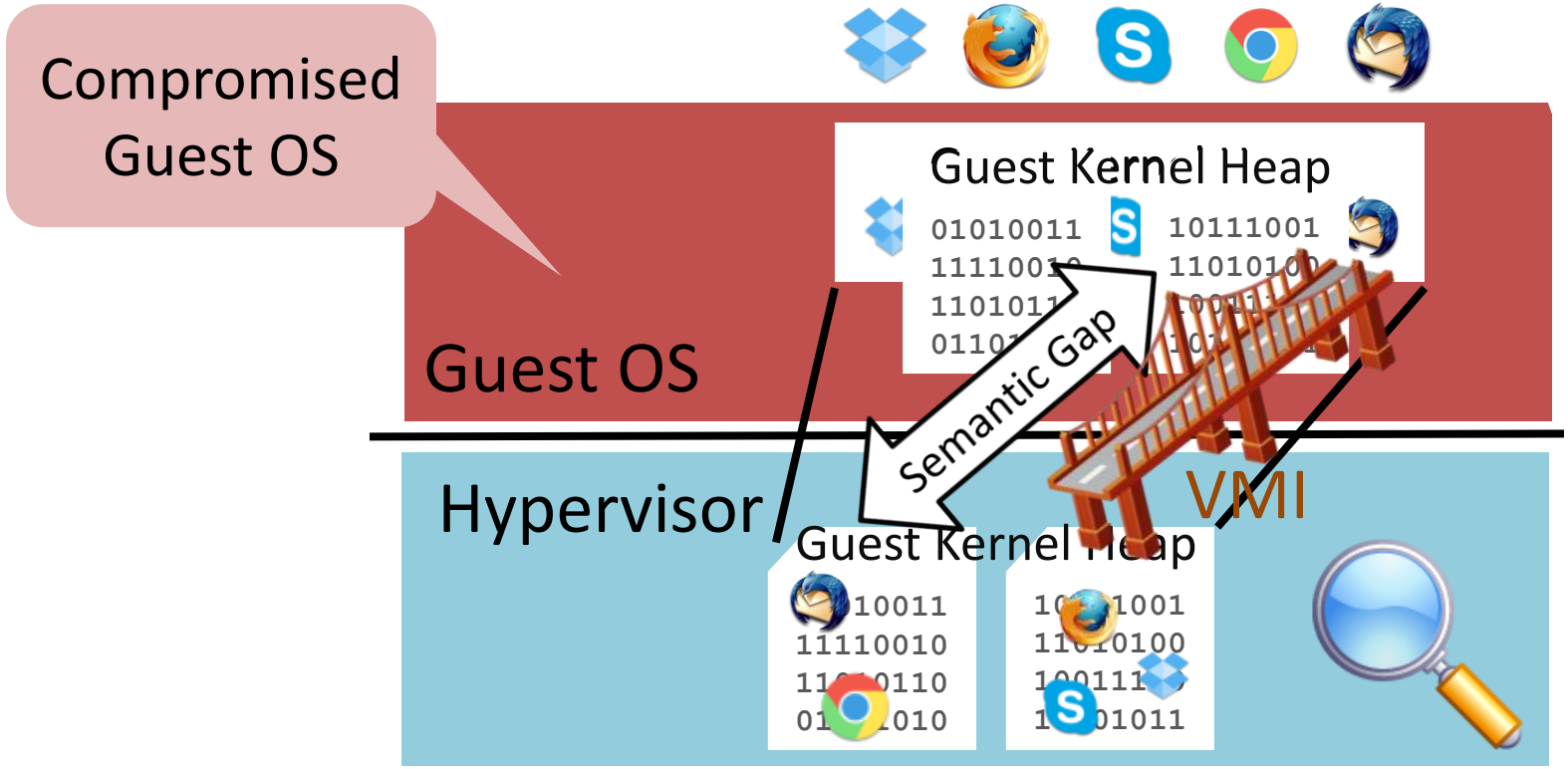


Semantic Gap: A challenge for VMI



VMI Challenge : Bridge the semantic gap

Semantic Gap: A challenge for VMI



VMI Challenge : Bridge the semantic gap even for compromised guest

VMI: Rootkit Detection Technique

- VMI is building block for layered security
  - Trusted hypervisor monitors less trusted guest
- **Common VMI goal**:
  - List processes in guest and identify malicious ones
- **Rootkit goal**:
  - Confuse VMI & hide malicious process

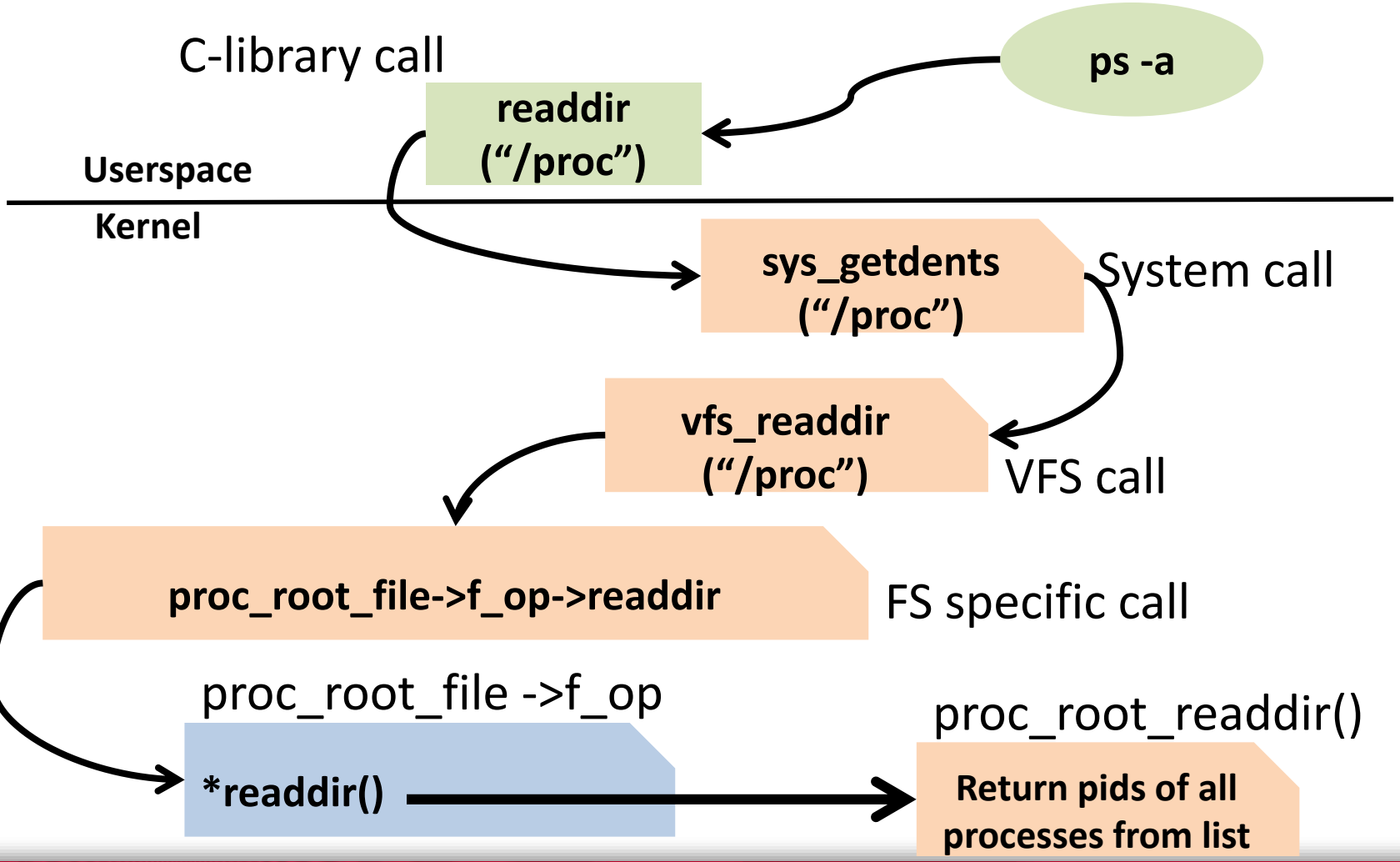
Rootkit Attack Techniques

- **Write text Segment**
  - .
- Kernel Object Hooking (KOH)
- Direct Kernel Object Manipulation (DKOM)
- Dynamic Kernel Structure Manipulation (DKSM)

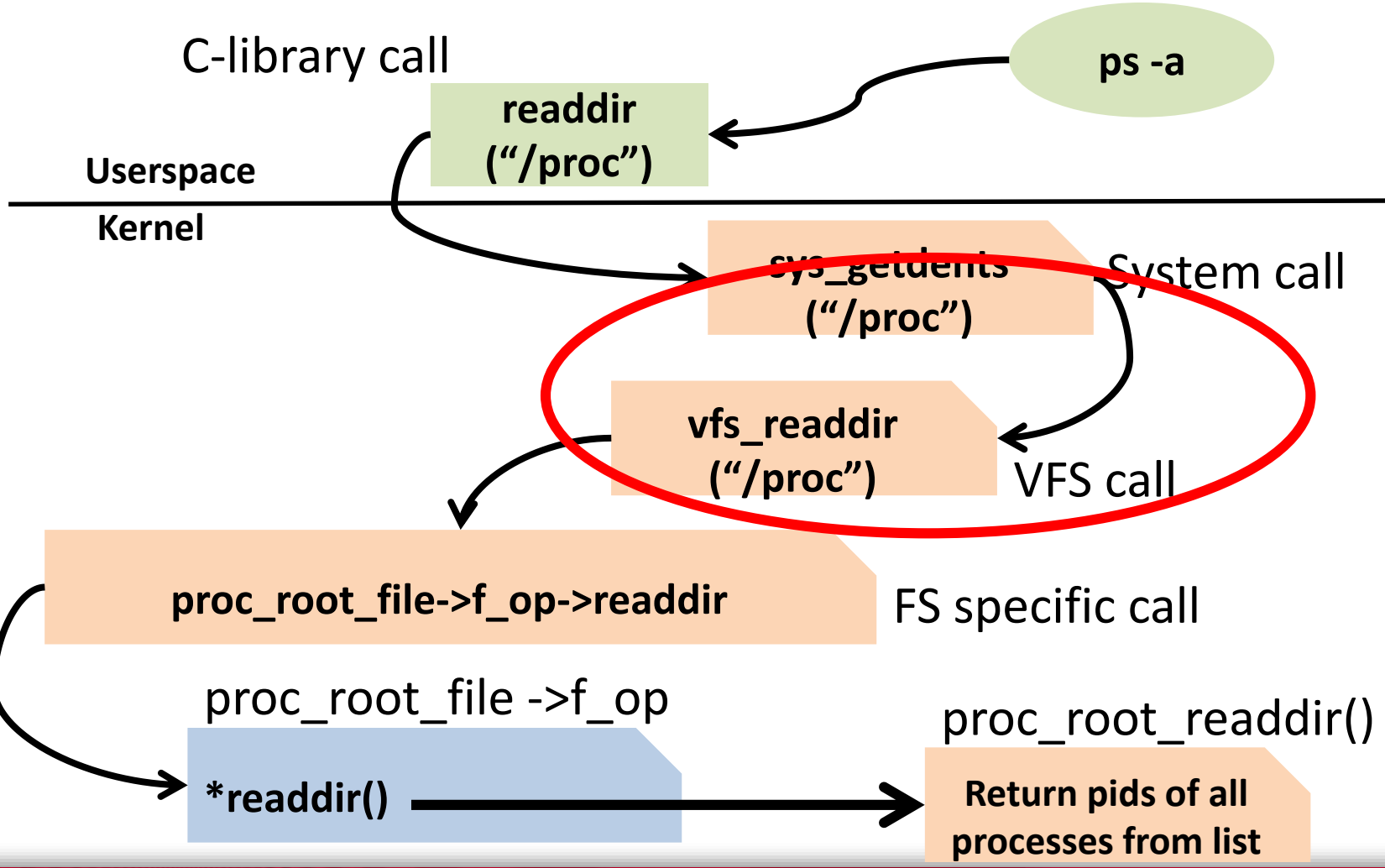
Rootkit Attack Techniques

- **Write text Segment**
  - Change `call` instruction argument
- **Kernel Object Hooking (KOH)**
- **Direct Kernel Object Manipulation (DKOM)**
- **Dynamic Kernel Structure Manipulation (DKSM)**

Listing all the running processes



Listing all the running processes



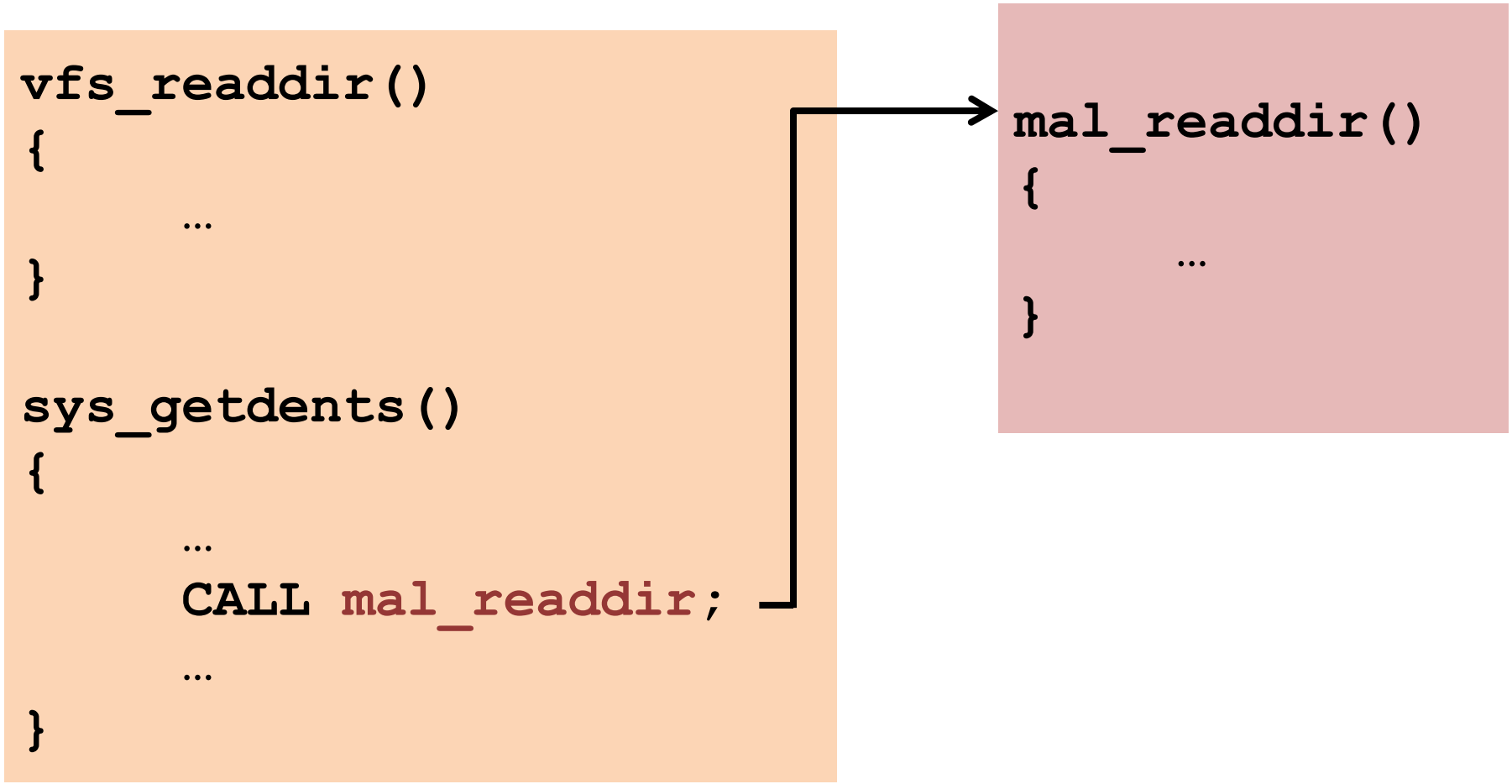
Change Text Section (1/2)

```
vfs_readdir()
{
        …
}

sys_getdents()
{
        …
        CALL vfs_readdir;
        …
}
```

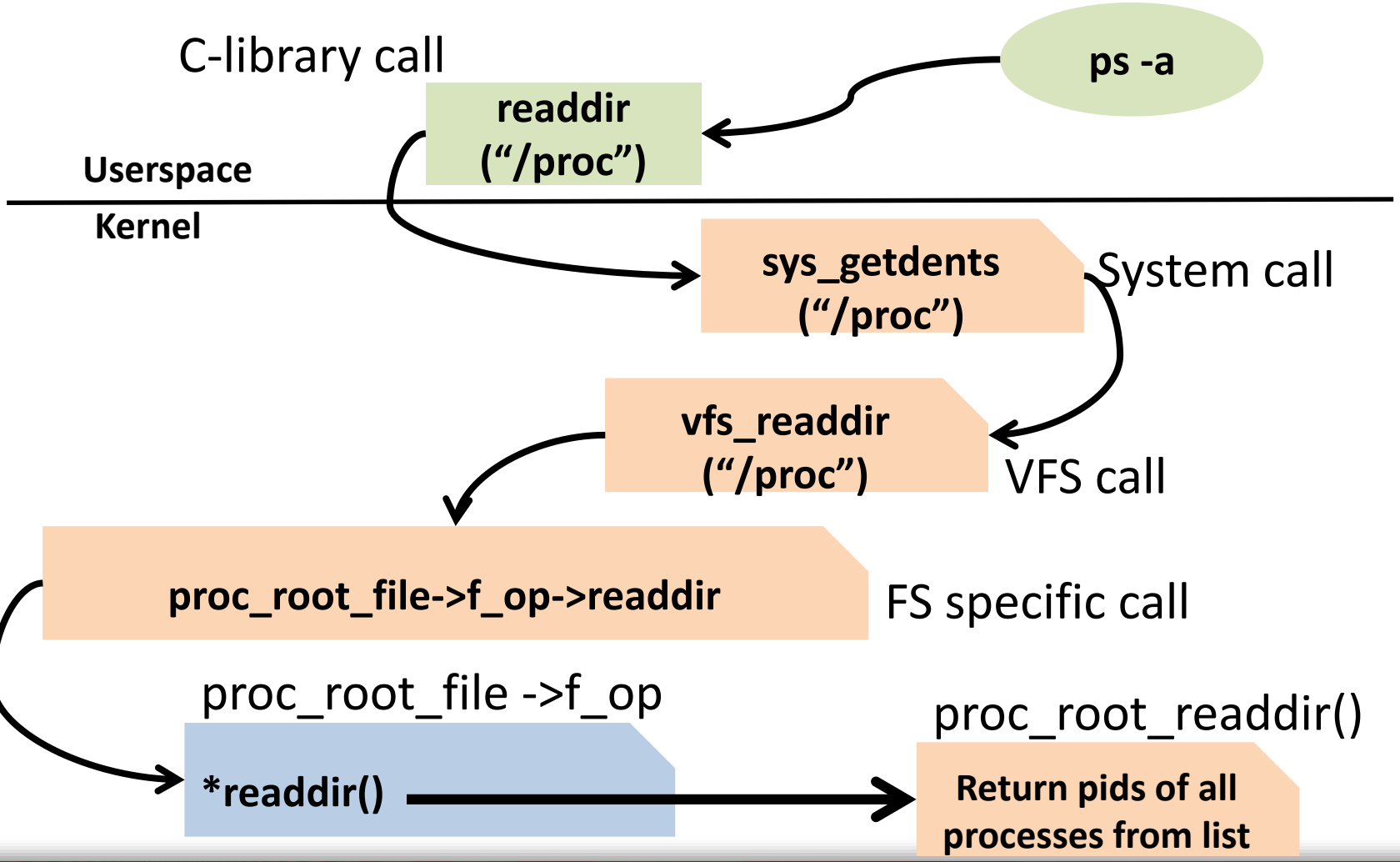
Change Text Section (1/2)

```
vfs_readdir()
{
        …
}

sys_getdents()
{
        …
        CALL mal_readdir;
        …
}
```

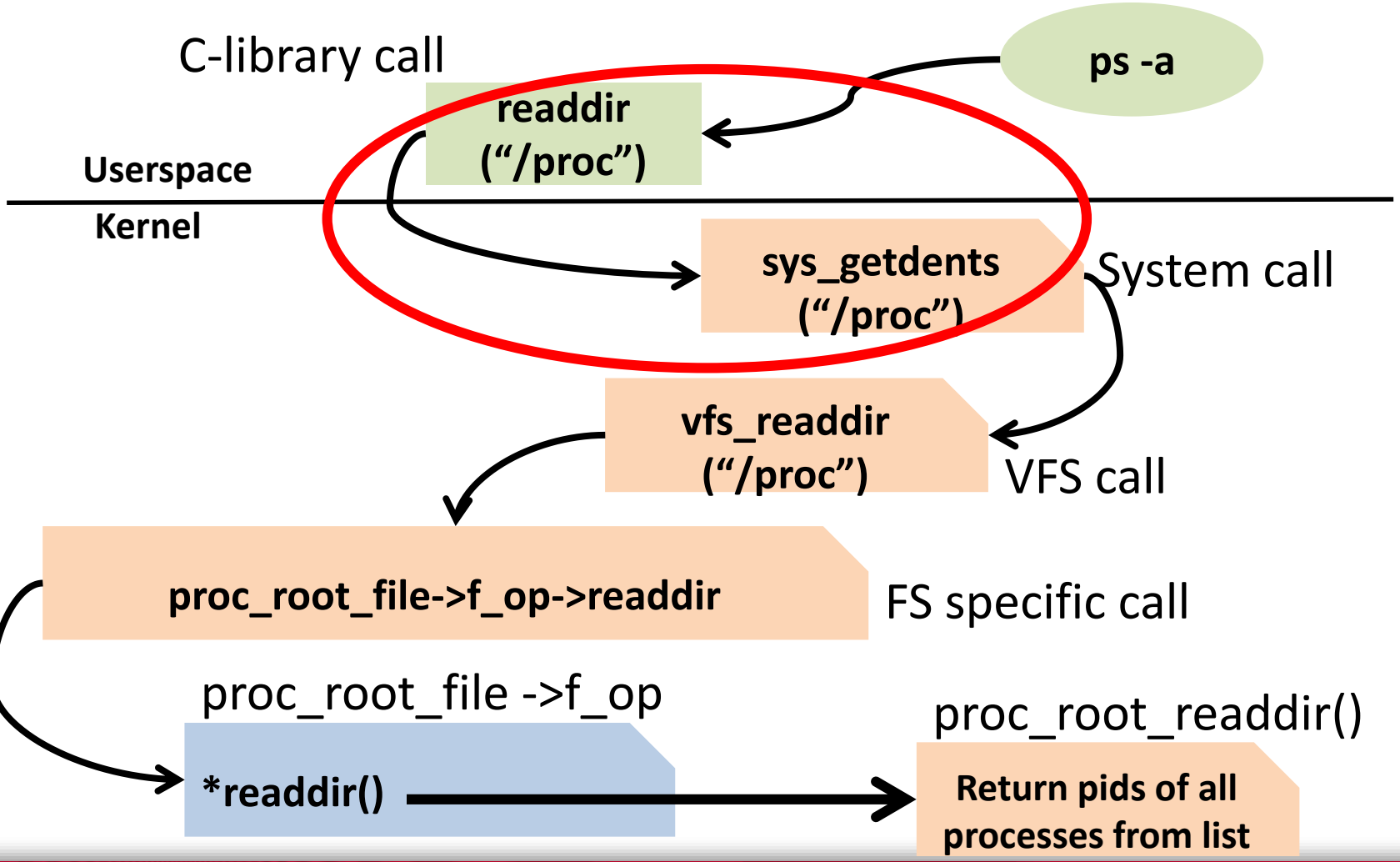
Change Text Section (1/2)



Listing all the running processes



Listing all the running processes

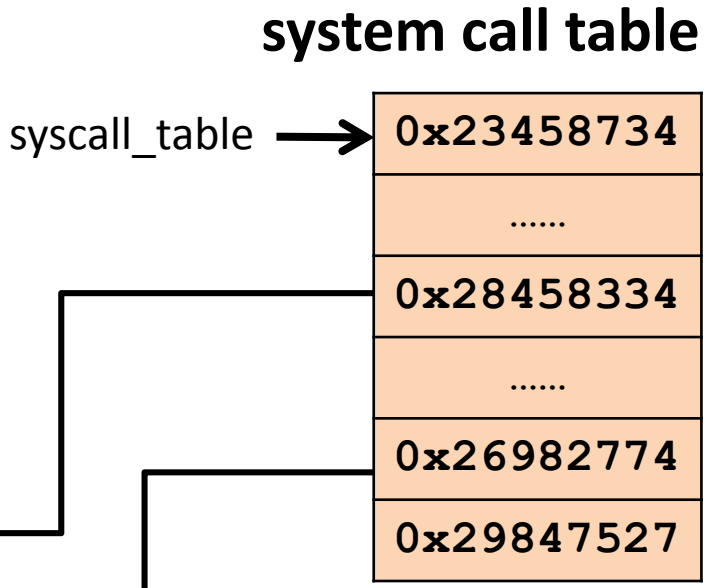


Change Text Section (2/2)

```
syscall_nr_getdents = 141
syscall_nr_open = 5
getdents:
CALL syscall_table[syscall_nr_getdents-1];
open:
CALL syscall_table[syscall_nr_open-1];
```

```
sys_open()
{
   …
}
```

```
sys_getdents()
{
   …
}
```

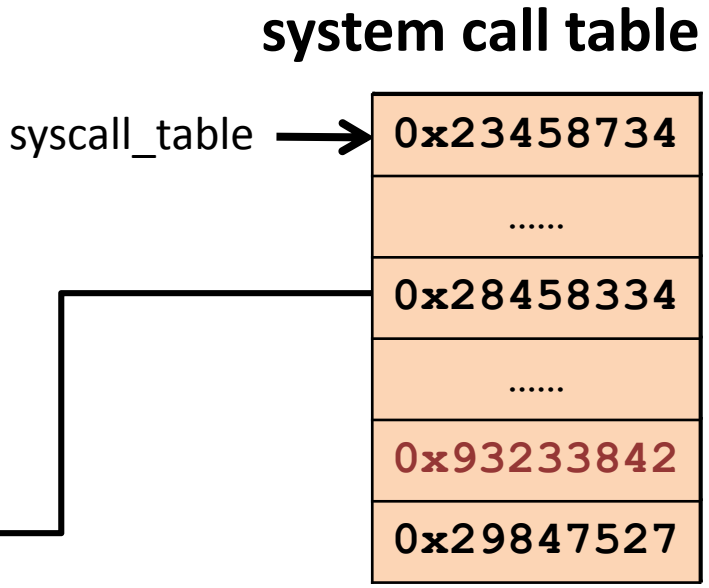


Change Text Section (2/2)

```
syscall_nr_getdents = 141
syscall_nr_open = 5
getdents:
CALL syscall_table[syscall_nr_getdents-1];
open:
CALL syscall_table[syscall_nr_open-1];
```

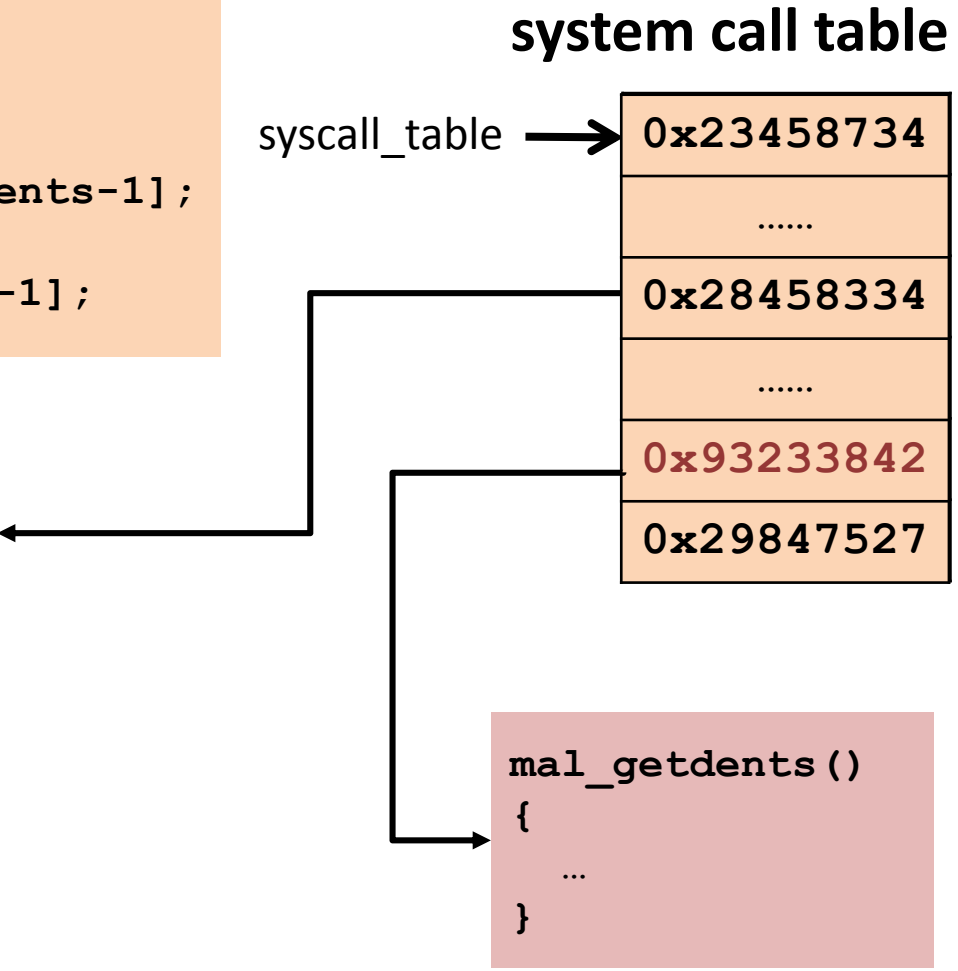
```
sys_open()
{
   …
}
```

```
sys_getdents()
{
   …
}
```



Change Text Section (2/2)

```
syscall_nr_getdents = 141
syscall_nr_open = 5
getdents:
CALL syscall_table[syscall_nr_getdents-1];
open:
CALL syscall_table[syscall_nr_open-1];
```



Write Text Segment Trust

- **Attack**: Change control flow by writing text segment
  -
-
  -
  -
  -
-

Write Text Segment Trust

- **Attack**: Change control flow by writing text segment
  - System call table or interrupt descriptor table
-
  -
  -
  -
-

Write Text Segment Trust

- **Attack**: Change control flow by writing text segment
  - System call table or interrupt descriptor table
- **Defense**: Hypervisor enforced W ⊕ X
  -
  -
  -
-

Write Text Segment Trust

- **Attack**: Change control flow by writing text segment
  - System call table or interrupt descriptor table
- **Defense**: Hypervisor enforced W ⊕ X
  - **W ⊕ X**: All pages writable or executable not both
  - Prevent guest from overwriting executable code pages
  - State of Art: SecVisor[8]
-

Write Text Segment Trust

- **Attack**: Change control flow by writing text segment
  - System call table or interrupt descriptor table
- **Defense**: Hypervisor enforced W ⊕ X
  - **W ⊕ X**: All pages writable or executable not both
  - Prevent guest from overwriting executable code pages
  - State of Art: SecVisor[8]
- **Trust Assumption**: Initial text segment benign

Write Text Segment Trust

- **Attack**: Change control flow by writing text segment
  - System call table or interrupt descriptor table
- **Defense**: Hypervisor enforced W ⊕ X
  - **W ⊕ X**: All pages writable or executable not both
  - Prevent guest from overwriting executable code pages
  - State of Art: SecVisor[8]
- **Trust Assumption**: Initial text segment benign

Cannot prevent attacks on control data in data segment

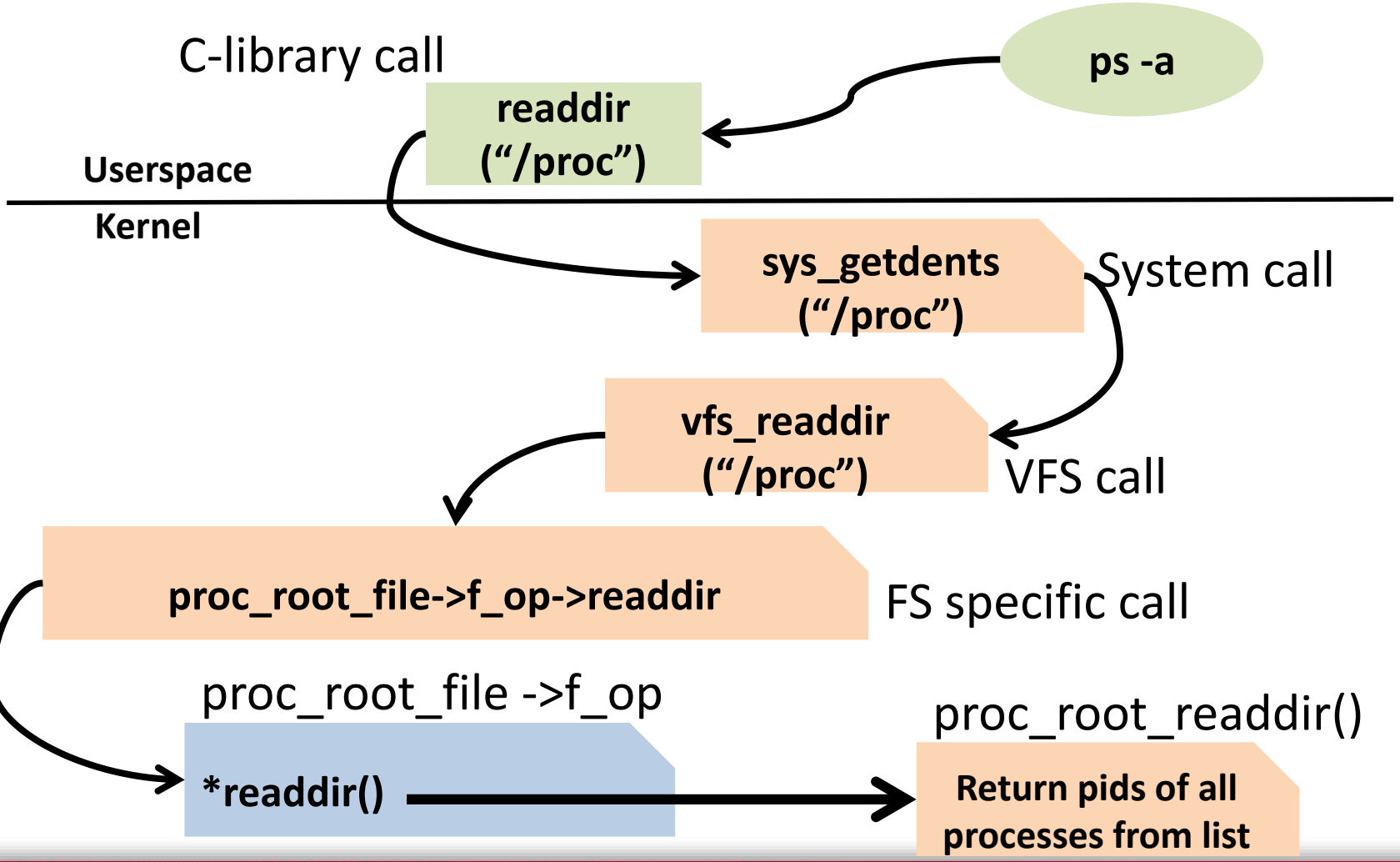
Rootkit Attack Techniques

- **Write text Segment**
  - Change `call` instruction argument
- **Kernel Object Hooking (KOH)**
  -
- **Direct Kernel Object Manipulation (DKOM)**
- **Dynamic Kernel Structure Manipulation (DKSM)**

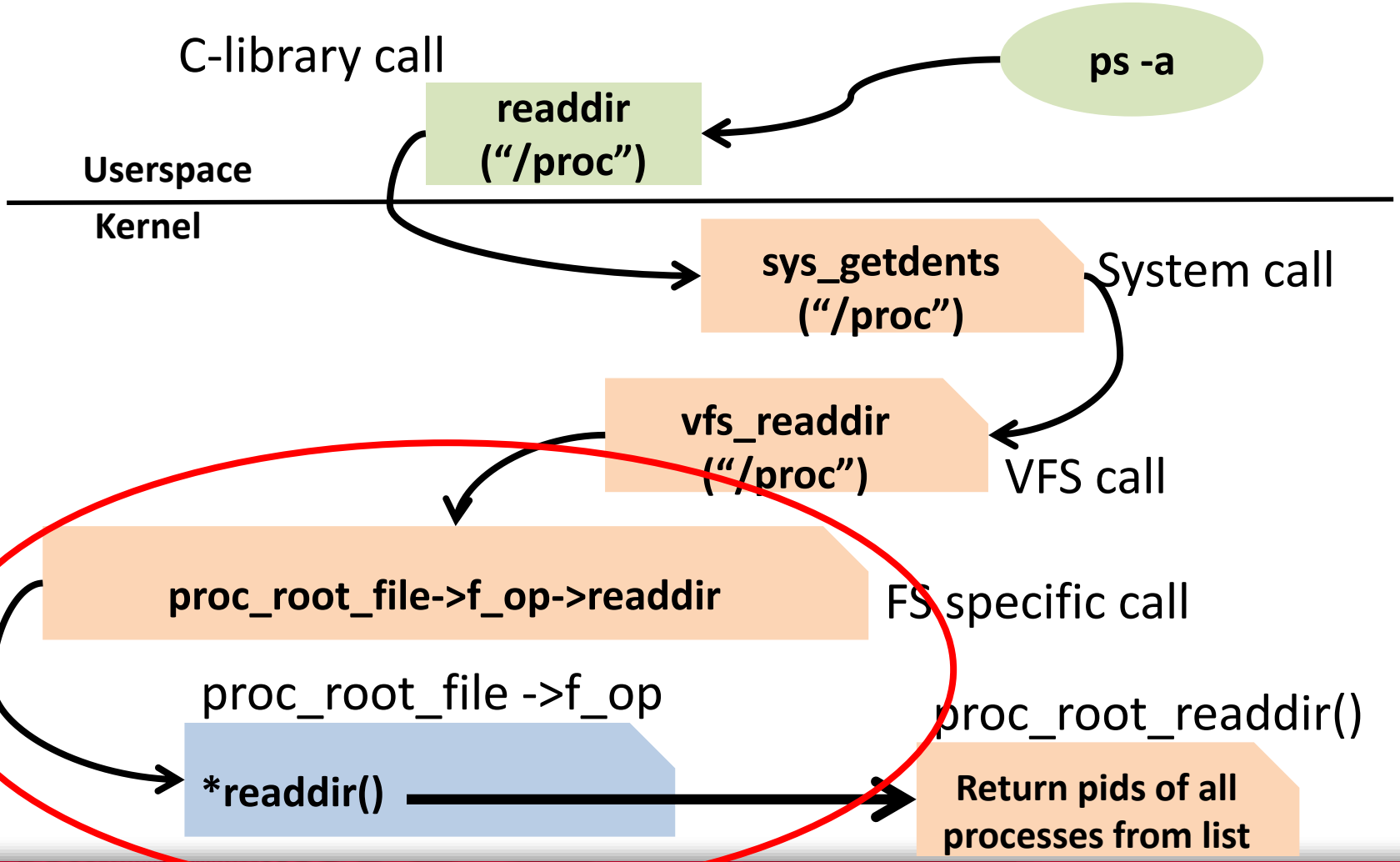
Rootkit Attack Techniques

- **Write text Segment**
  - Change `call` instruction argument
- **Kernel Object Hooking (KOH)**
  - Change function pointers (data segment)
- **Direct Kernel Object Manipulation (DKOM)**
- **Dynamic Kernel Structure Manipulation (DKSM)**

Listing all the running processes



Listing all the running processes



Kernel Object Hooking (KOH)

```
struct file{
   …
   struct path f_path;
   const struct file_operations
        *f_op;
   struct fown_struct f_owner;
   void *f_security;
   fmode_t f_mode;
   off_t f_pos;
   …
}
```

file struct object for /proc

```
struct file_operations{
   …
   int (*open) (struct inode
*, struct file *);
   ssize_t (*read) (struct
file *, char __user *,
size_t, loff_t *);
   int (*readdir) (struct
file *, void *, filldir_t);
   …
}
```

file_operations object for /proc

```
proc_root_readdir(){
   …
}
```

readdir code for proc filesystem

Kernel Object Hooking (KOH)

```
struct file{
   …
   struct path f_path;
   const struct file_operations
        *f_op;
   struct fown_struct f_owner;
   void *f_security;
   fmode_t f_mode;
   off_t f_pos;
   …
}
```

file struct object for /proc

```
struct file_operations{
   …
   int (*open) (struct inode
*, struct file *);
   ssize_t (*read) (struct
file *, char __user *,
size_t, loff_t *);
   int (*readdir) (struct
file *, void *, filldir_t);
  …
}
```

file_operations object for /proc

```
proc_root_readdir(){
   …
}
```

readdir code for proc filesystem

Kernel Object Hooking (KOH)

```
struct file{
   …
   struct path f_path;
   const struct file_operations
        *f_op;
   struct fown_struct f_owner;
   void *f_security;
   fmode_t f_mode;
   off_t f_pos;
   …
}
```

file struct object for /proc

```
struct file_operations{
   …
   int (*open) (struct inode
*, struct file *);
   ssize_t (*read) (struct
file *, char __user *,
size_t, loff_t *);
   int (*readdir) (struct
file *, void *, filldir_t);
   …
}
```

file_operations object for /proc

```
proc_root_readdir(){
   …
}
```

readdir code for proc filesystem

```
mal_readdir(){
   …
}
```

KOH Defense Trust Assumptions

- **Attack**: Change function pointers in objects
- **Defense**: Protect initialized function pointers
  - Redirect hooks to write protected memory
  - State of Art: HookSafe[9]
- **Trust Assumption**: Pristine initial OS copy
  - Administrator can white-list safe modules
  - All hooks are learned during dynamic analysis

Cannot protect data fields in heap section

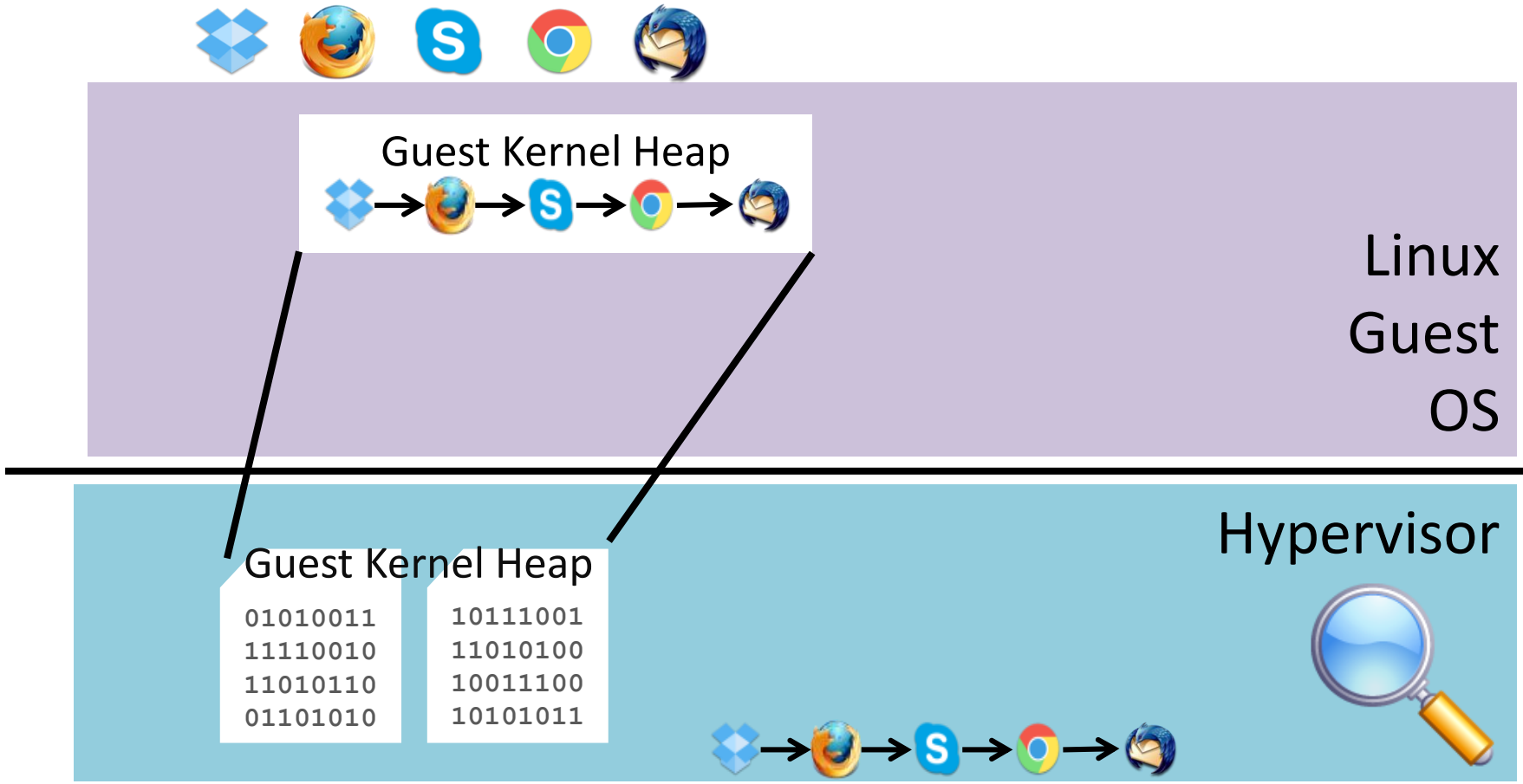
Rootkit Attack Techniques

- **Write text Segment**
  - Change `call` instruction argument
- **Kernel Object Hooking (KOH)**
  - Change function pointers (data segment)
- **Direct Kernel Object Manipulation (DKOM)**
  -
- **Dynamic Kernel Structure Manipulation (DKSM)**

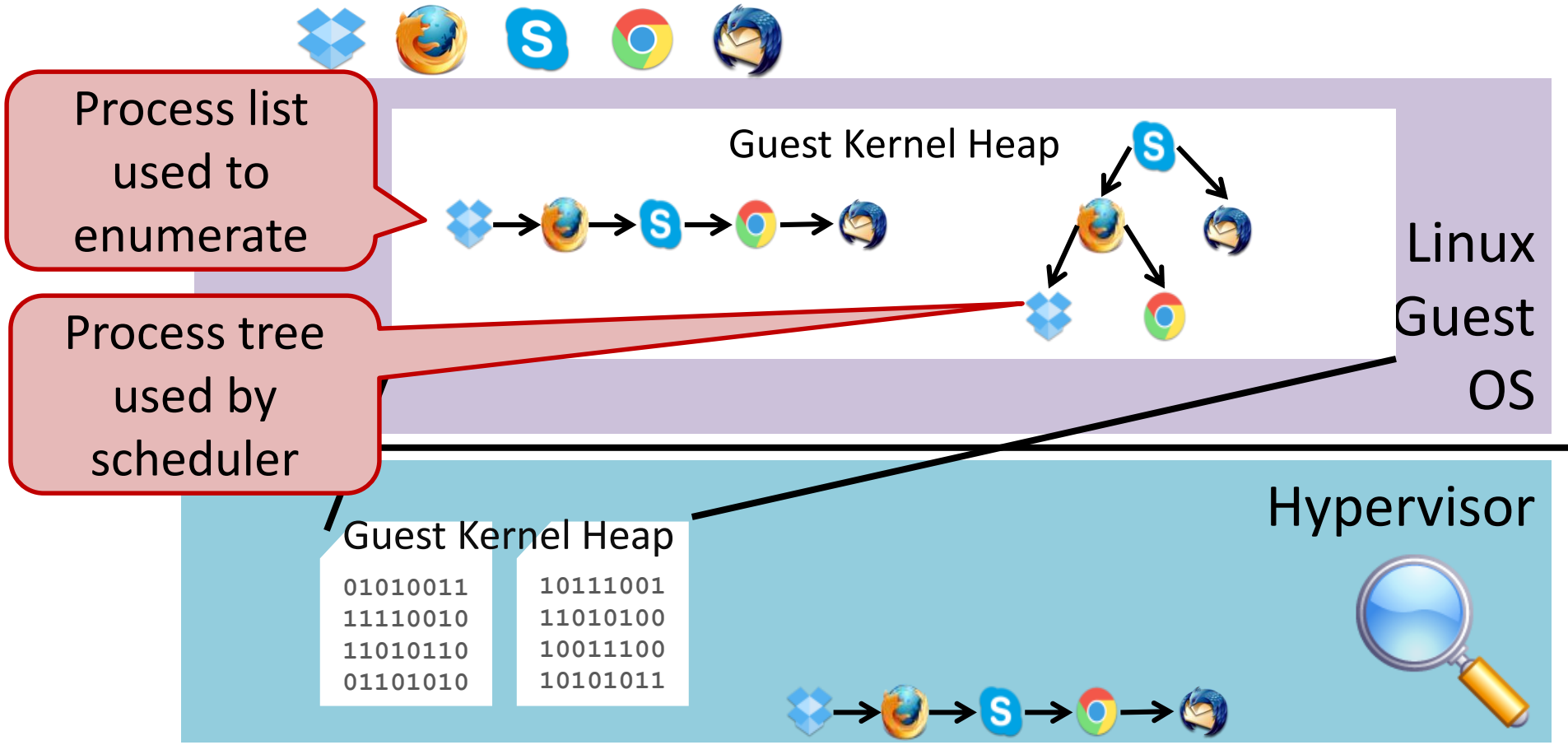
Rootkit Attack Techniques

- **Write text Segment**
  - Change `call` instruction argument
- **Kernel Object Hooking (KOH)**
  - Change function pointers (data segment)
- **Direct Kernel Object Manipulation (DKOM)**
  - Manipulate heap objects – violate invariants
- **Dynamic Kernel Structure Manipulation (DKSM)**

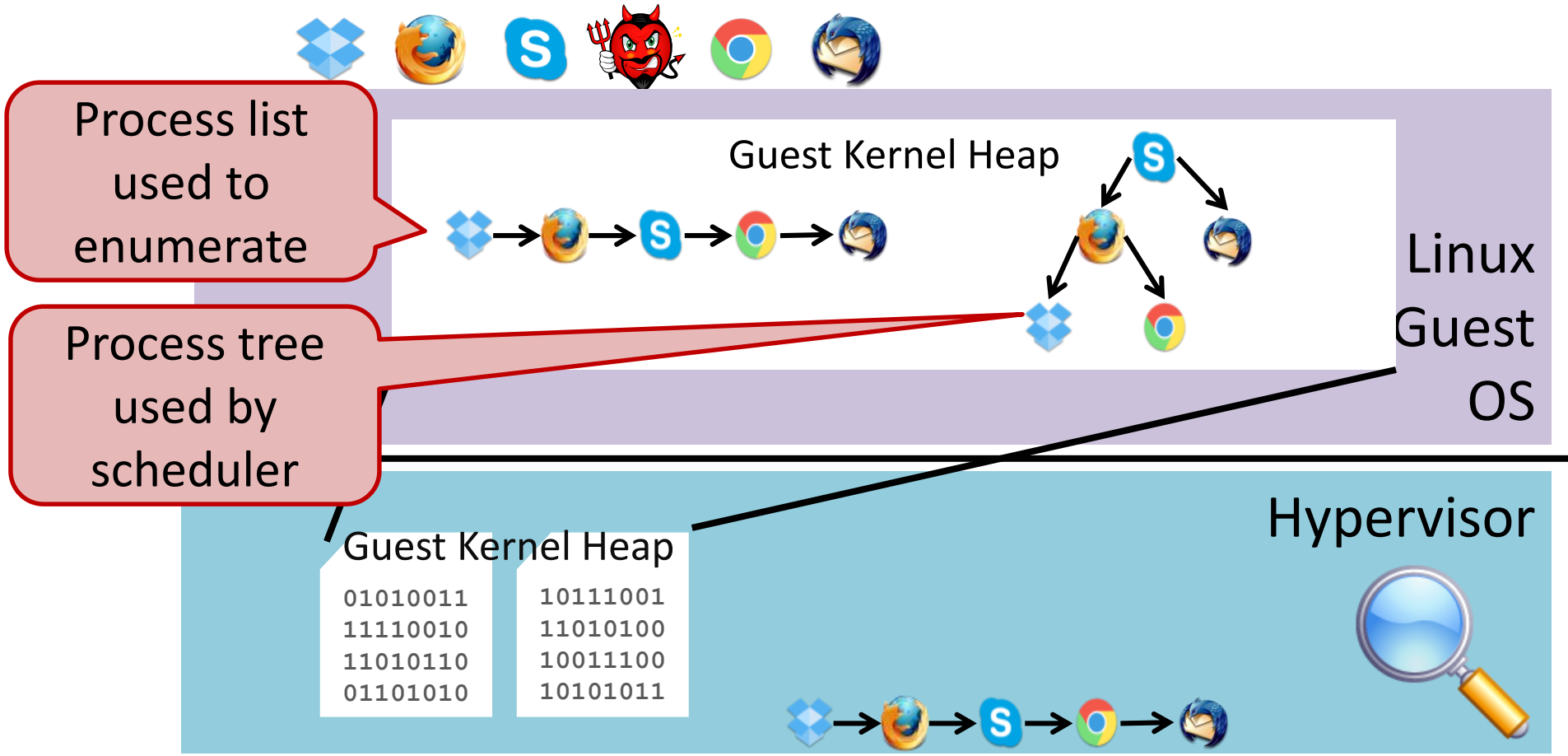
Direct Kernel Object Manipulation



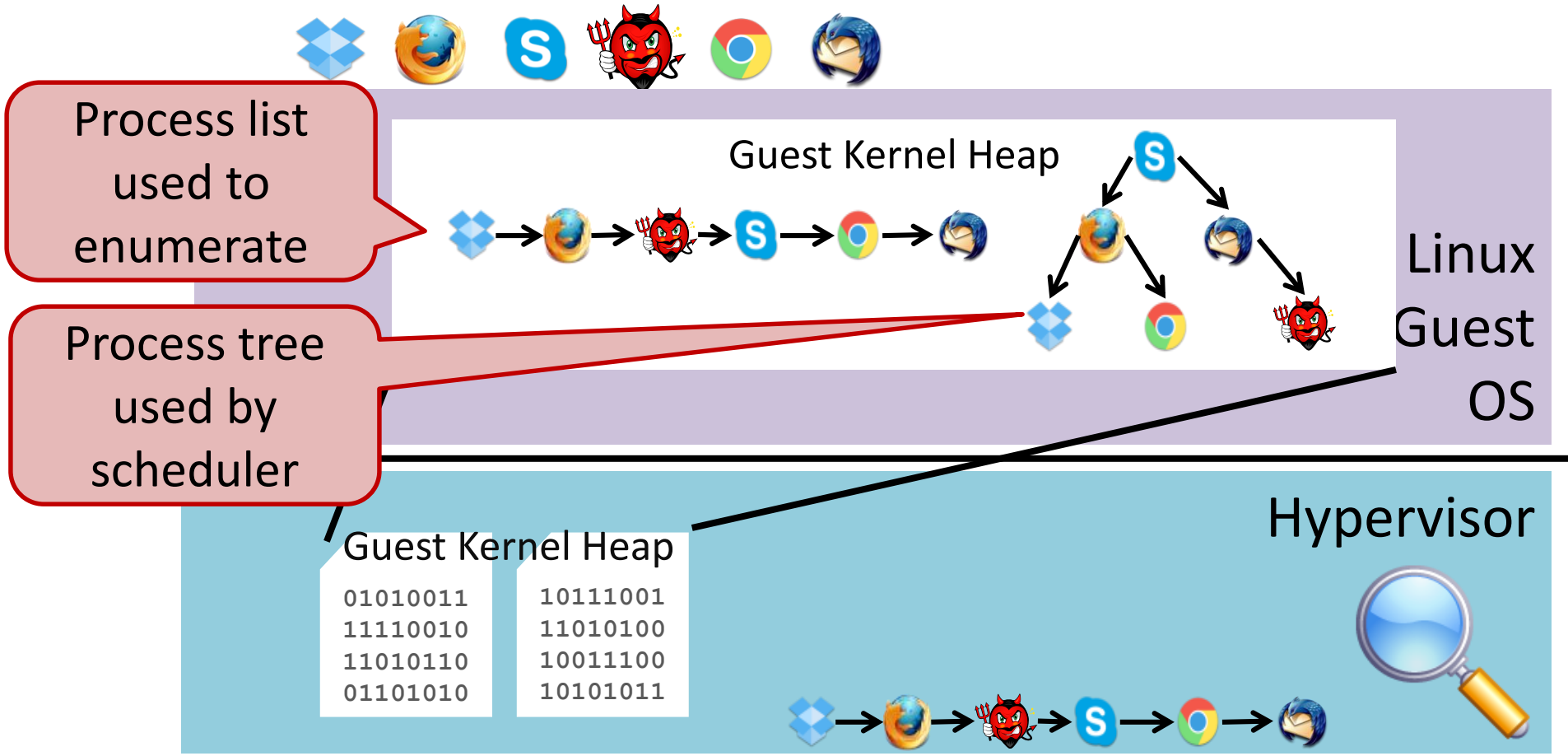
Direct Kernel Object Manipulation



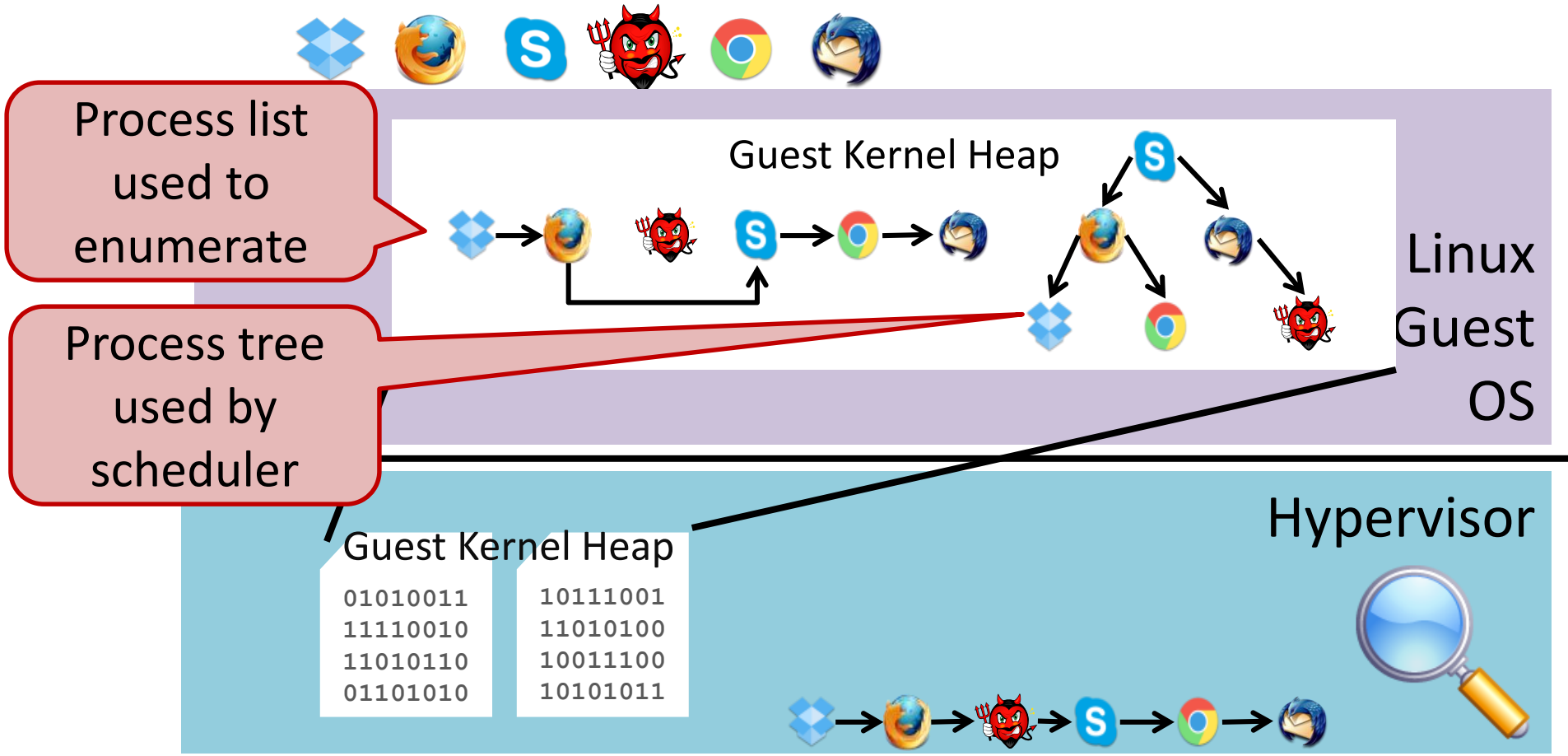
Direct Kernel Object Manipulation



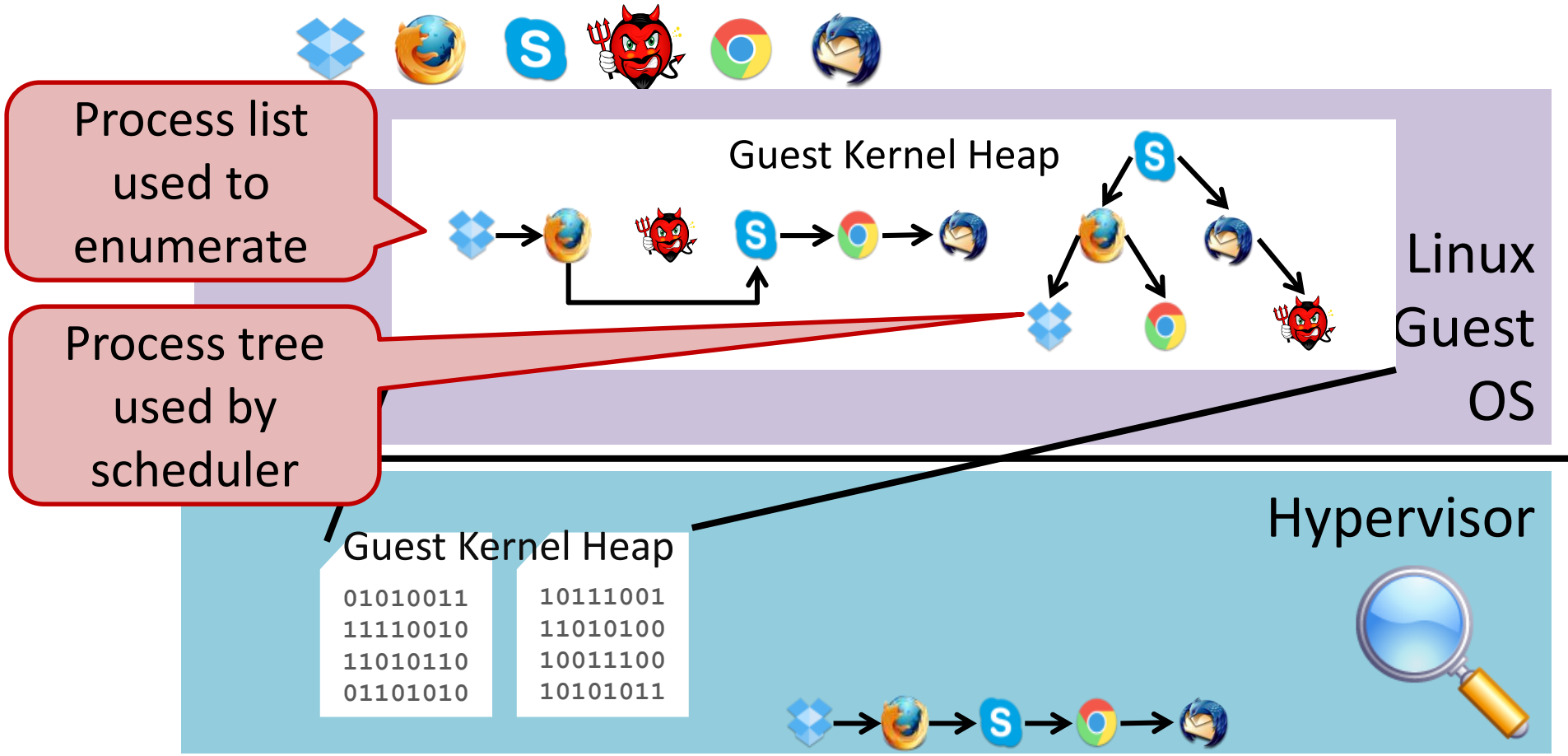
Direct Kernel Object Manipulation



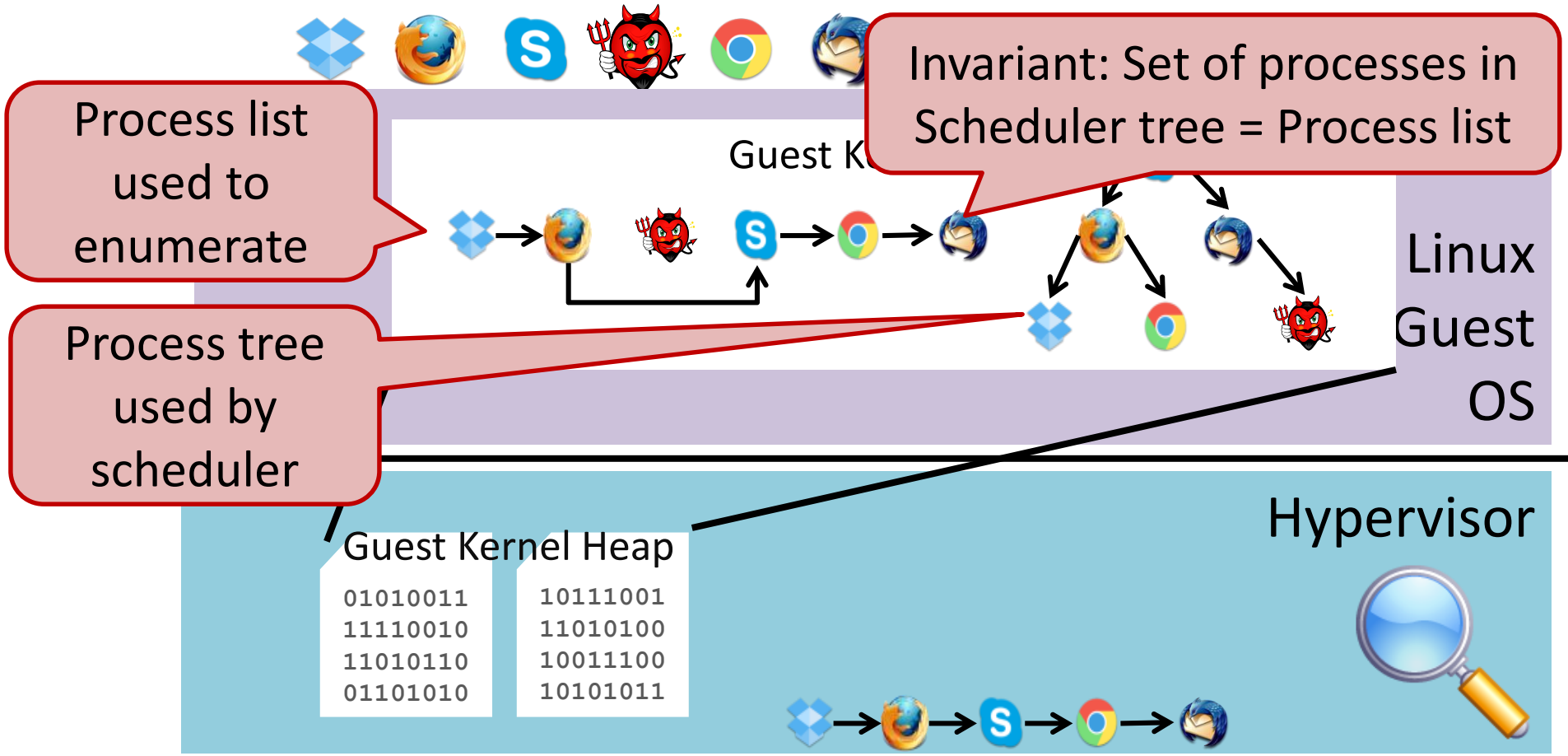
Direct Kernel Object Manipulation



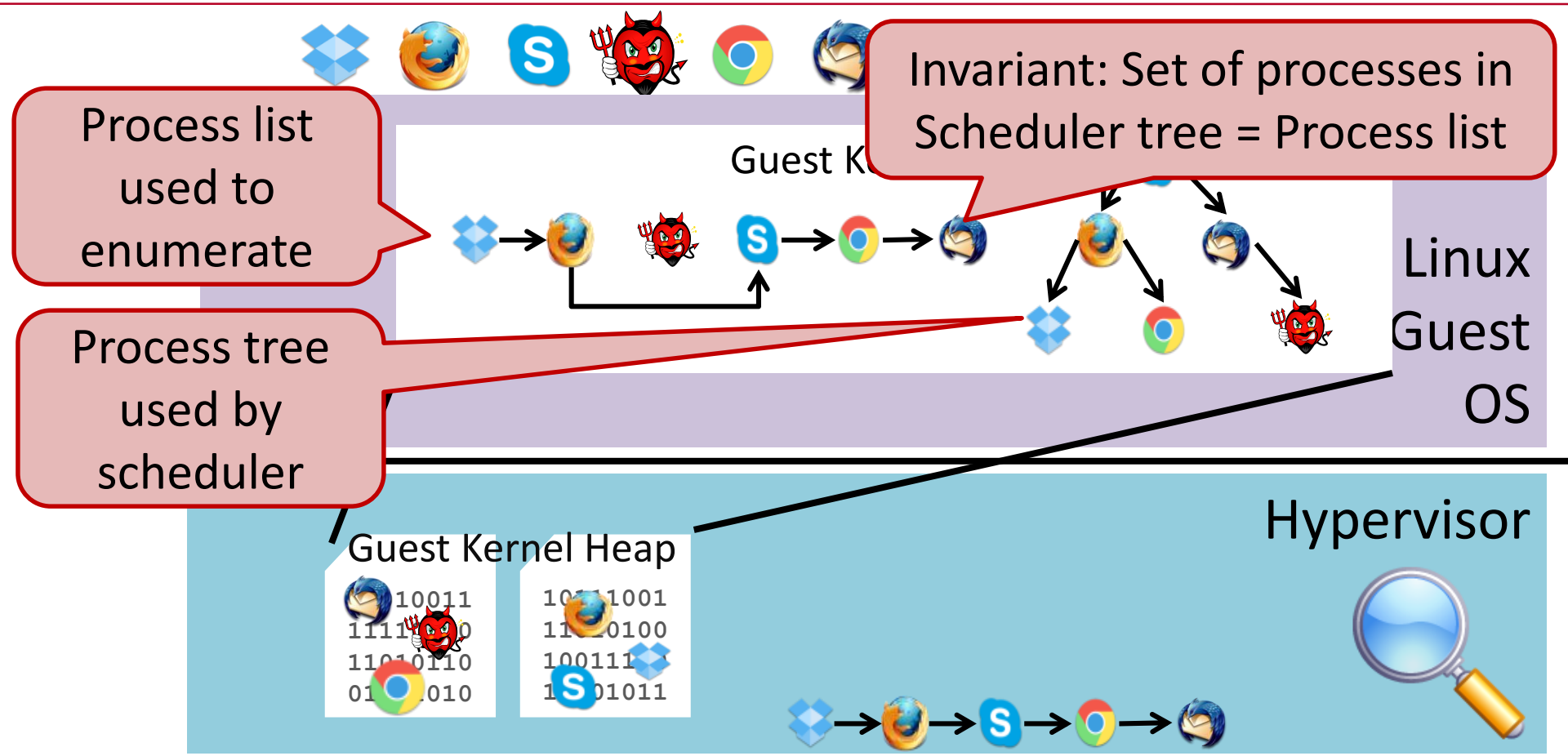
Direct Kernel Object Manipulation



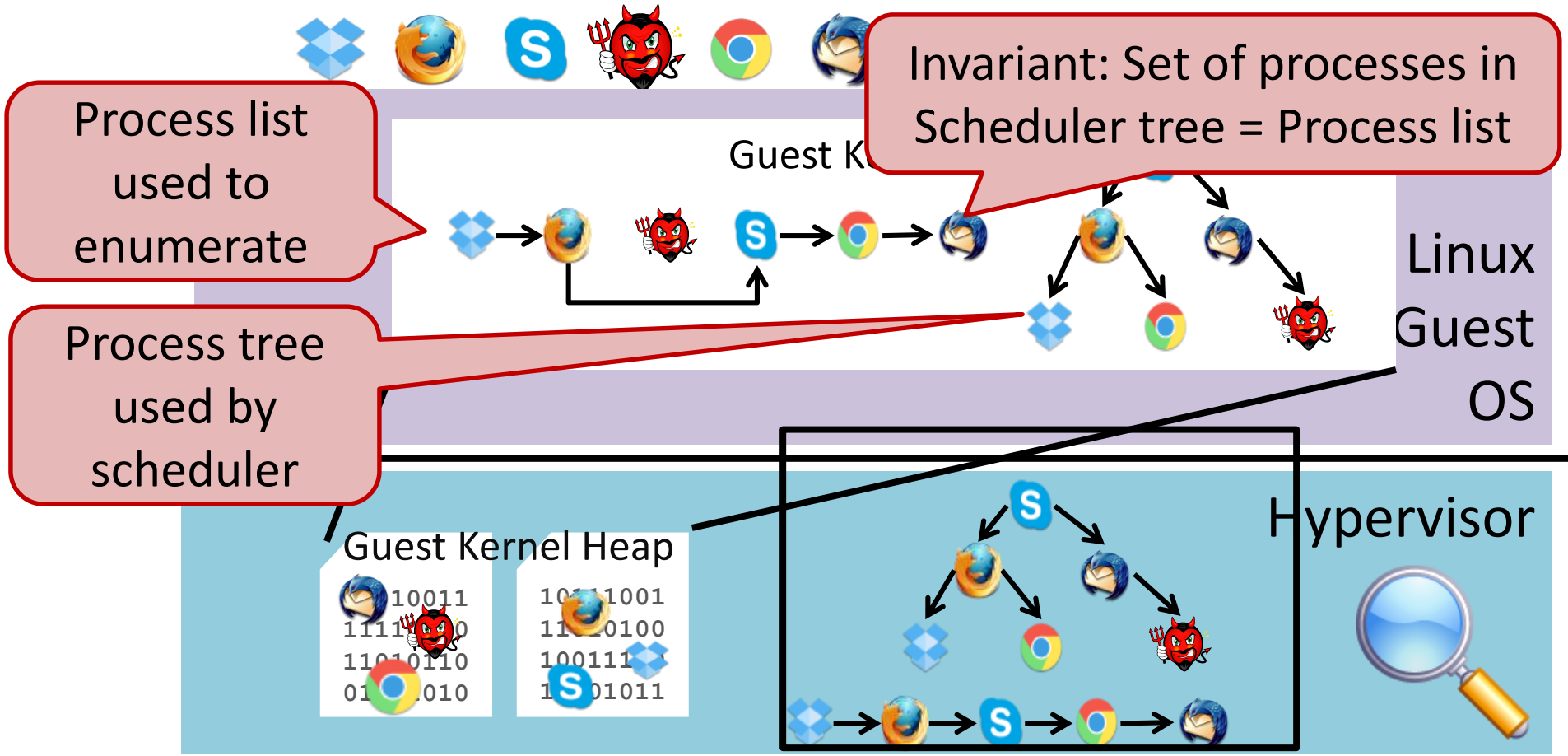
Direct Kernel Object Manipulation



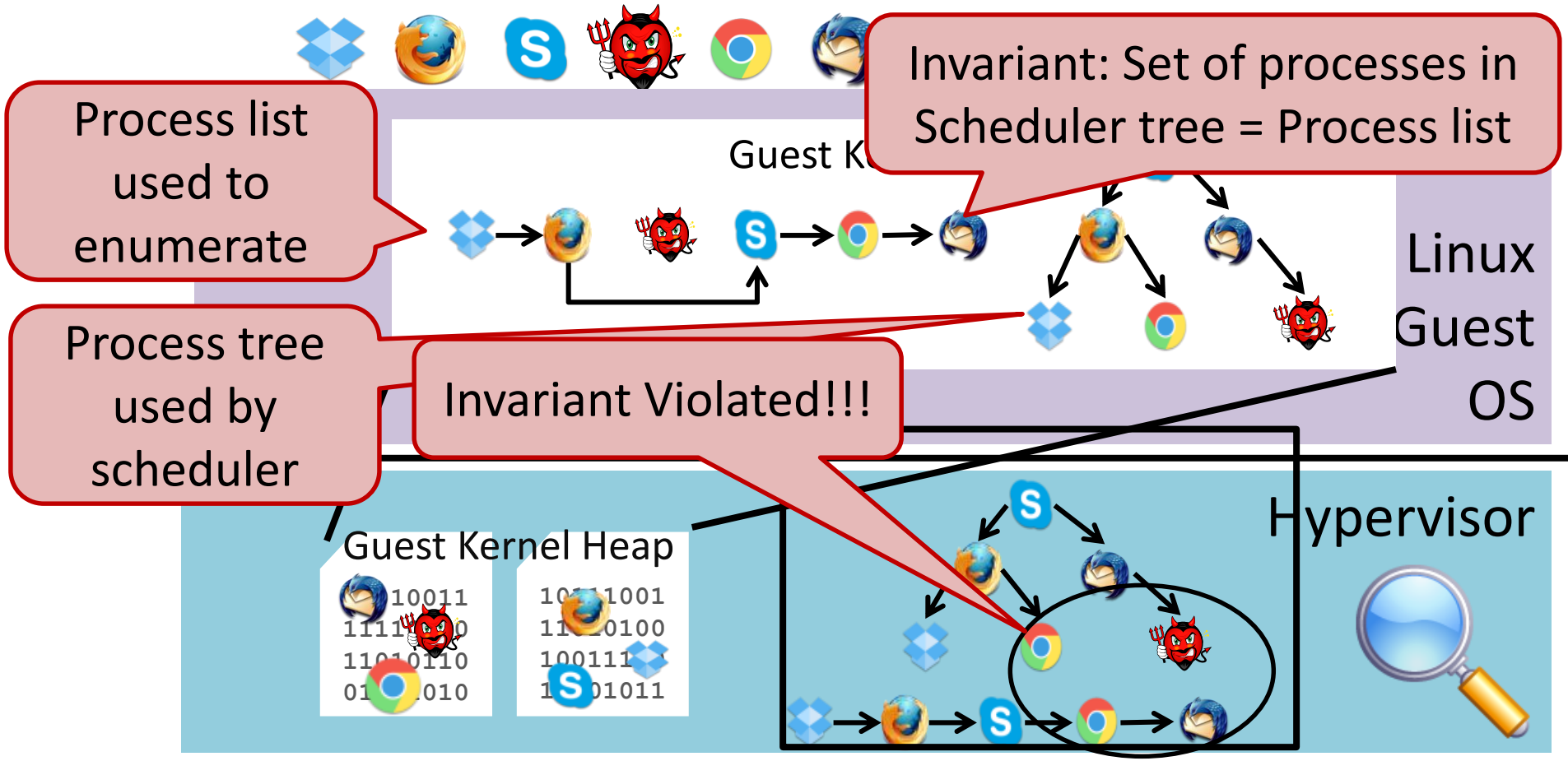
Direct Kernel Object Manipulation



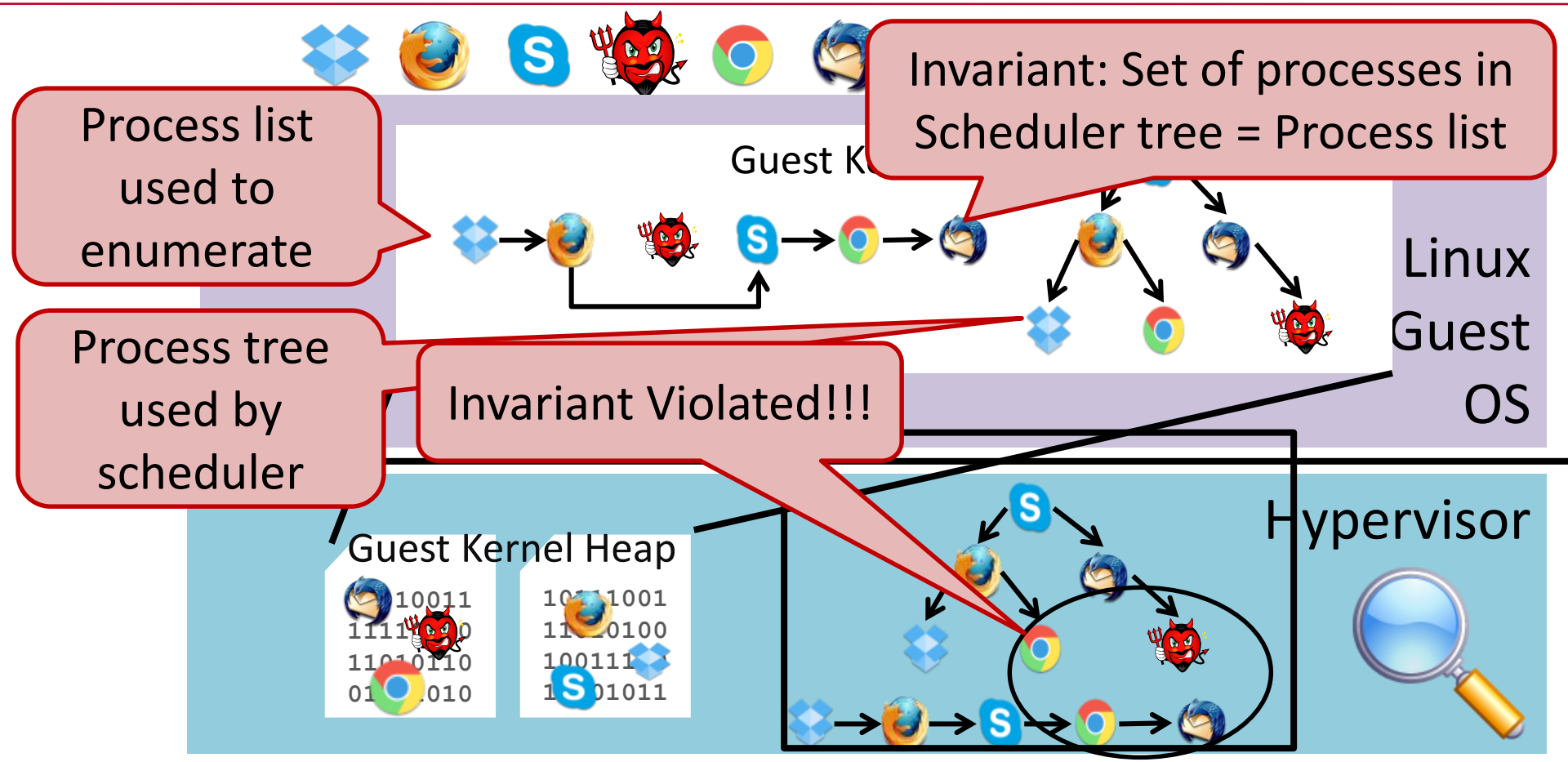
Direct Kernel Object Manipulation



Direct Kernel Object Manipulation



Direct Kernel Object Manipulation



Assume attacker can't win race with asynchronous checking

DKOM Defense Trust Assumptions

- **Attack**: Modify the kernel objects in heap
- **Defense**: Asynchronously check invariants
  - State of Art: OSck[10]
- **Trust Assumption**:
  - All security invariants can be learned
  - Invariants can be checked in single search
  - Attackers cannot win races with the monitor
  - Availability of other integrity defenses

Cannot prevent attacks or detect transient attacks

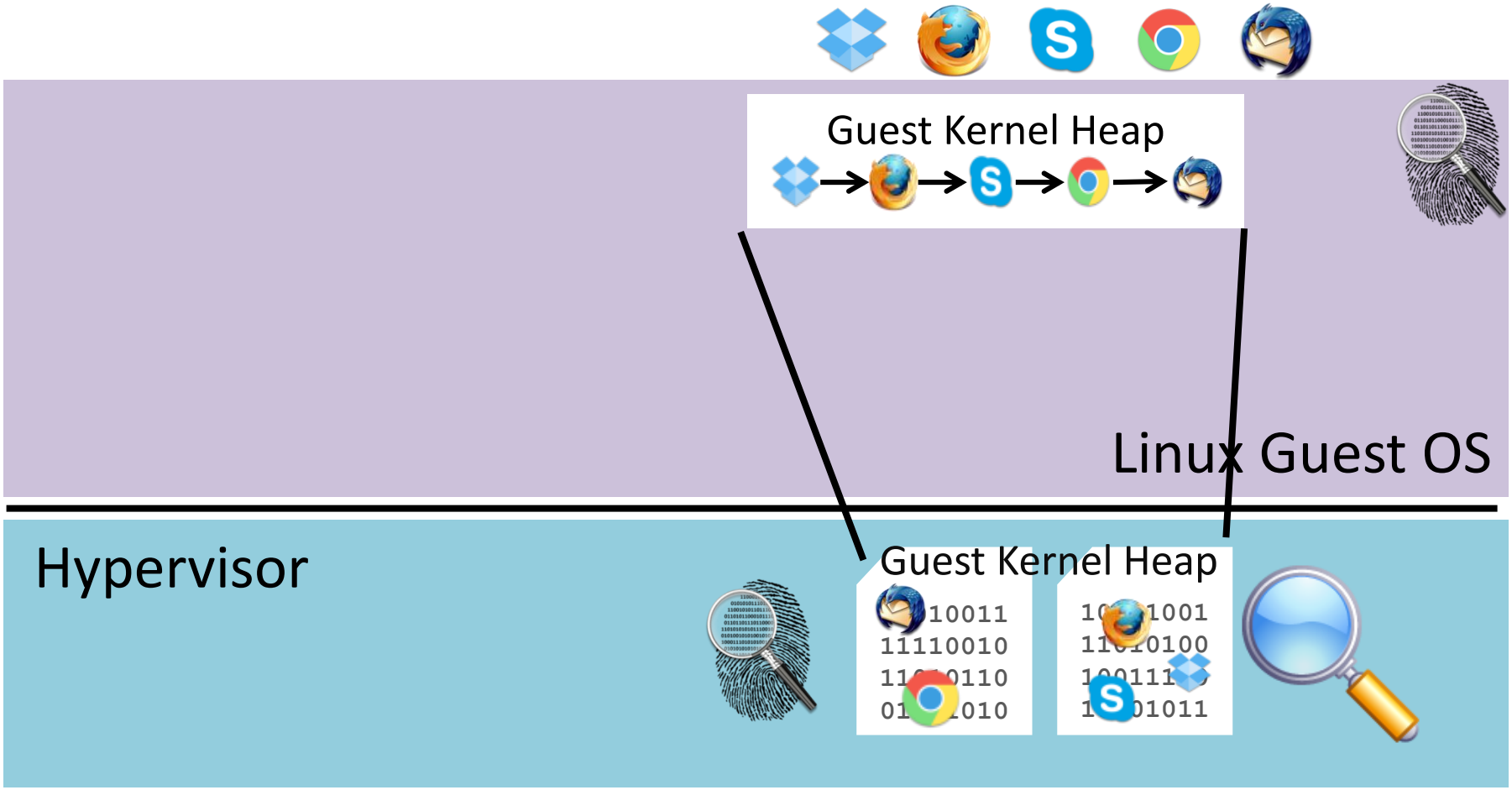
Rootkit Attack Techniques

- **Write text Segment**
  - Change `call` instruction argument
- **Kernel Object Hooking (KOH)**
  - Change function pointers (data segment)
- **Direct Kernel Object Manipulation (DKOM)**
  - Manipulate heap objects – violate invariants
- **Dynamic Kernel Structure Manipulation (DKSM)**
  -

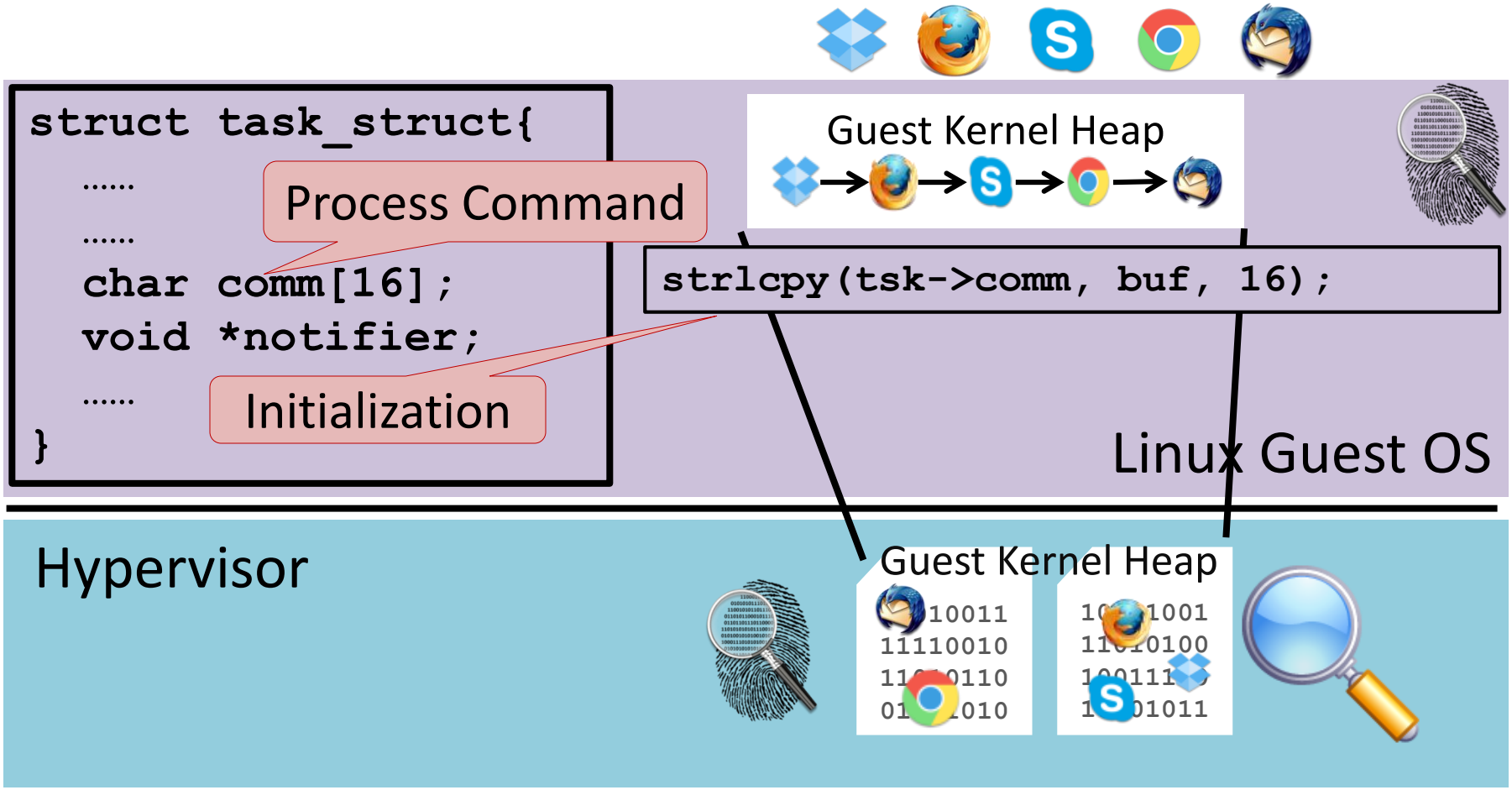
Rootkit Attack Techniques

- **Write text Segment**
  - Change `call` instruction argument
- **Kernel Object Hooking (KOH)**
  - Change function pointers (data segment)
- **Direct Kernel Object Manipulation (DKOM)**
  - Manipulate heap objects – violate invariants
- **Dynamic Kernel Structure Manipulation (DKSM)**
  - Change data structure interpretation

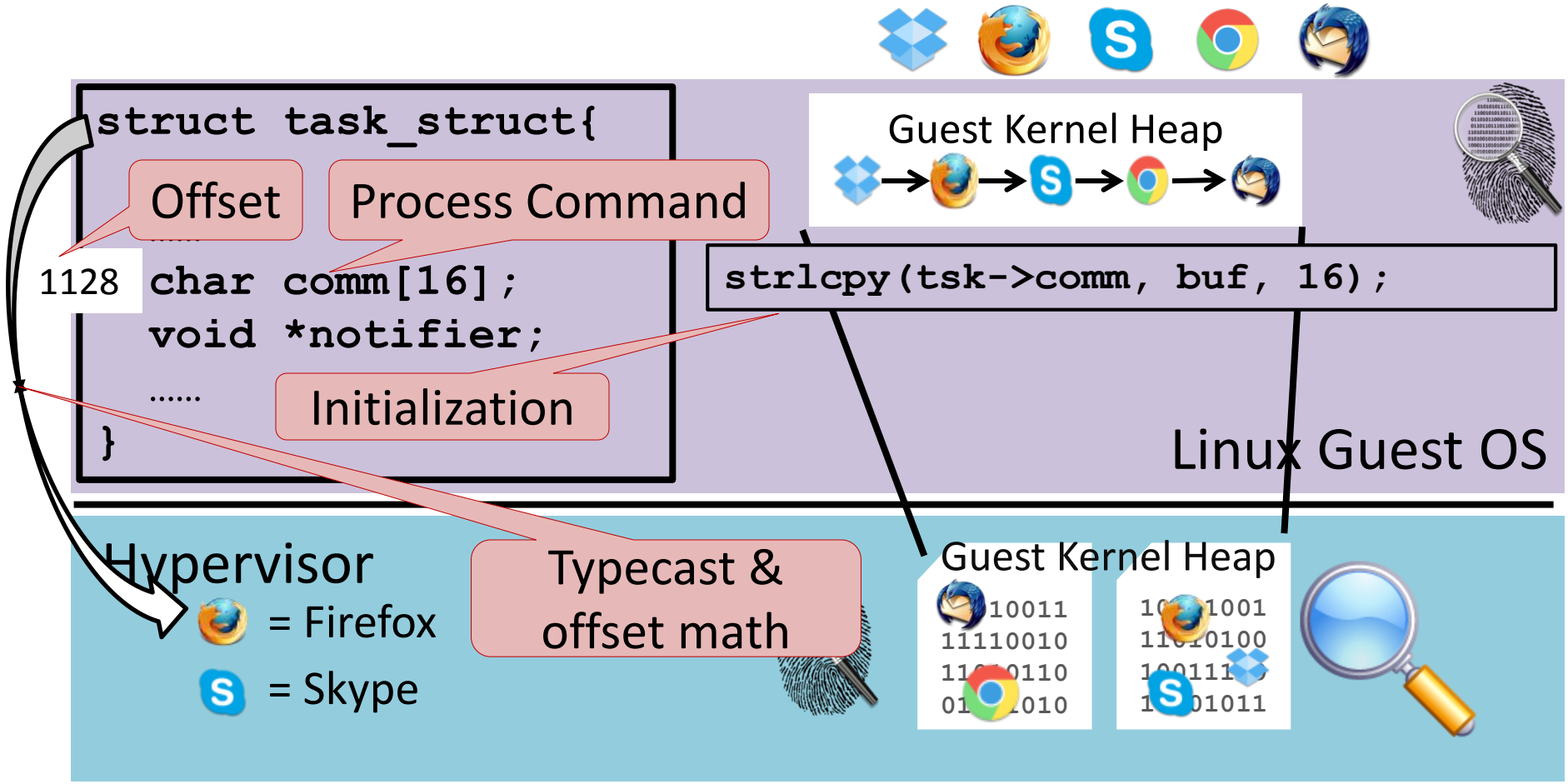
Data Structure Manipulation Attack



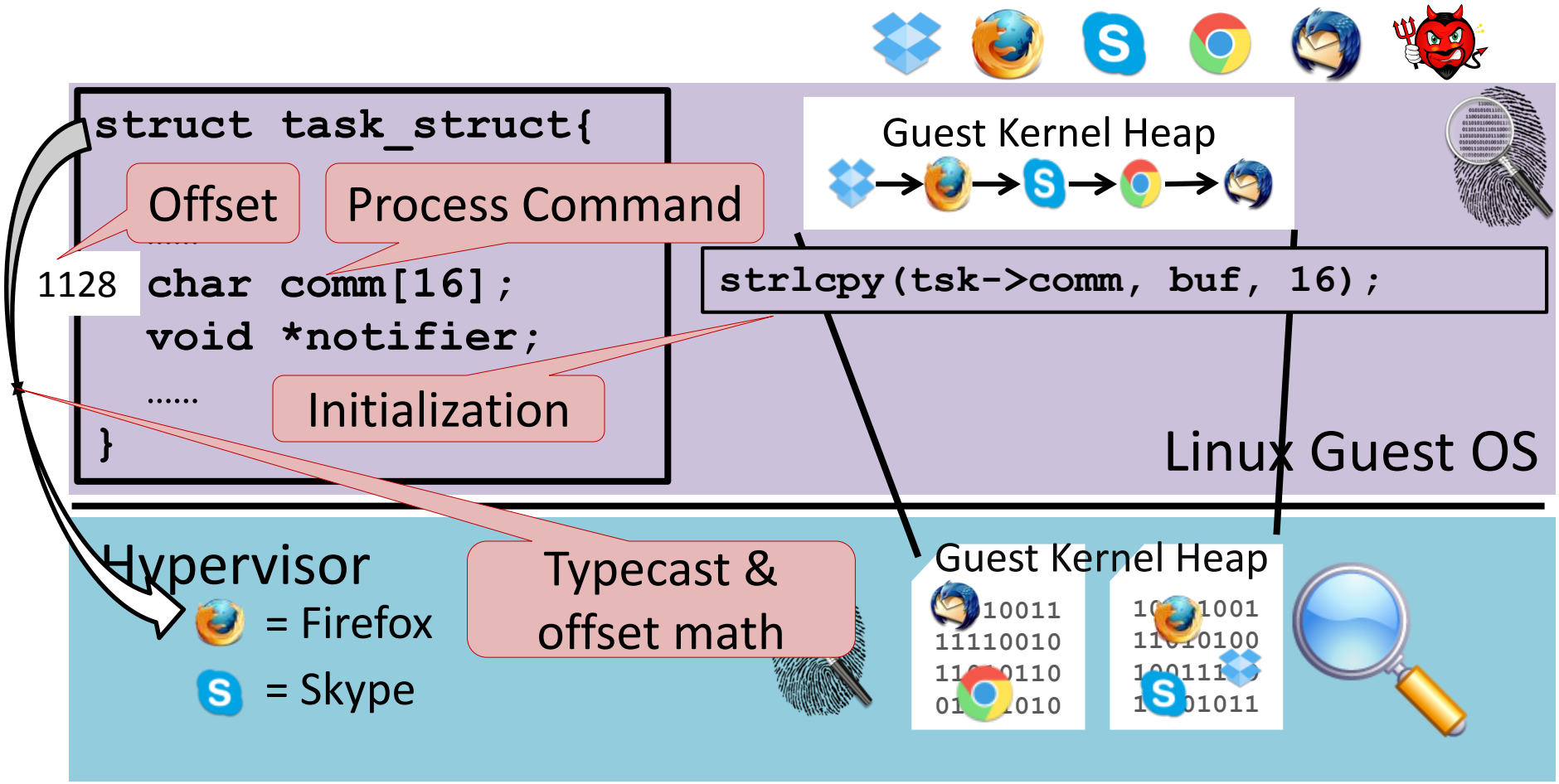
Data Structure Manipulation Attack



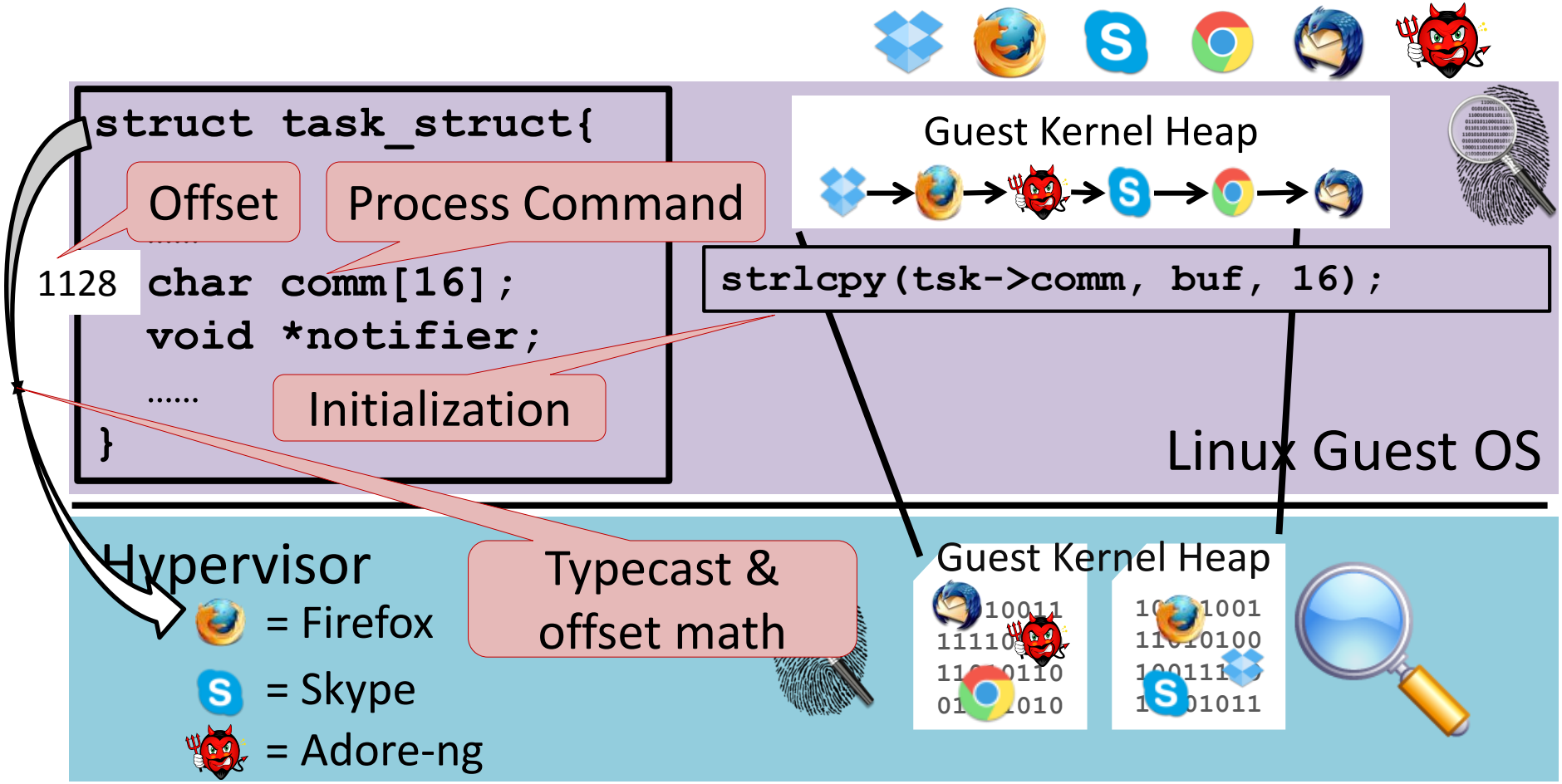
Data Structure Manipulation Attack



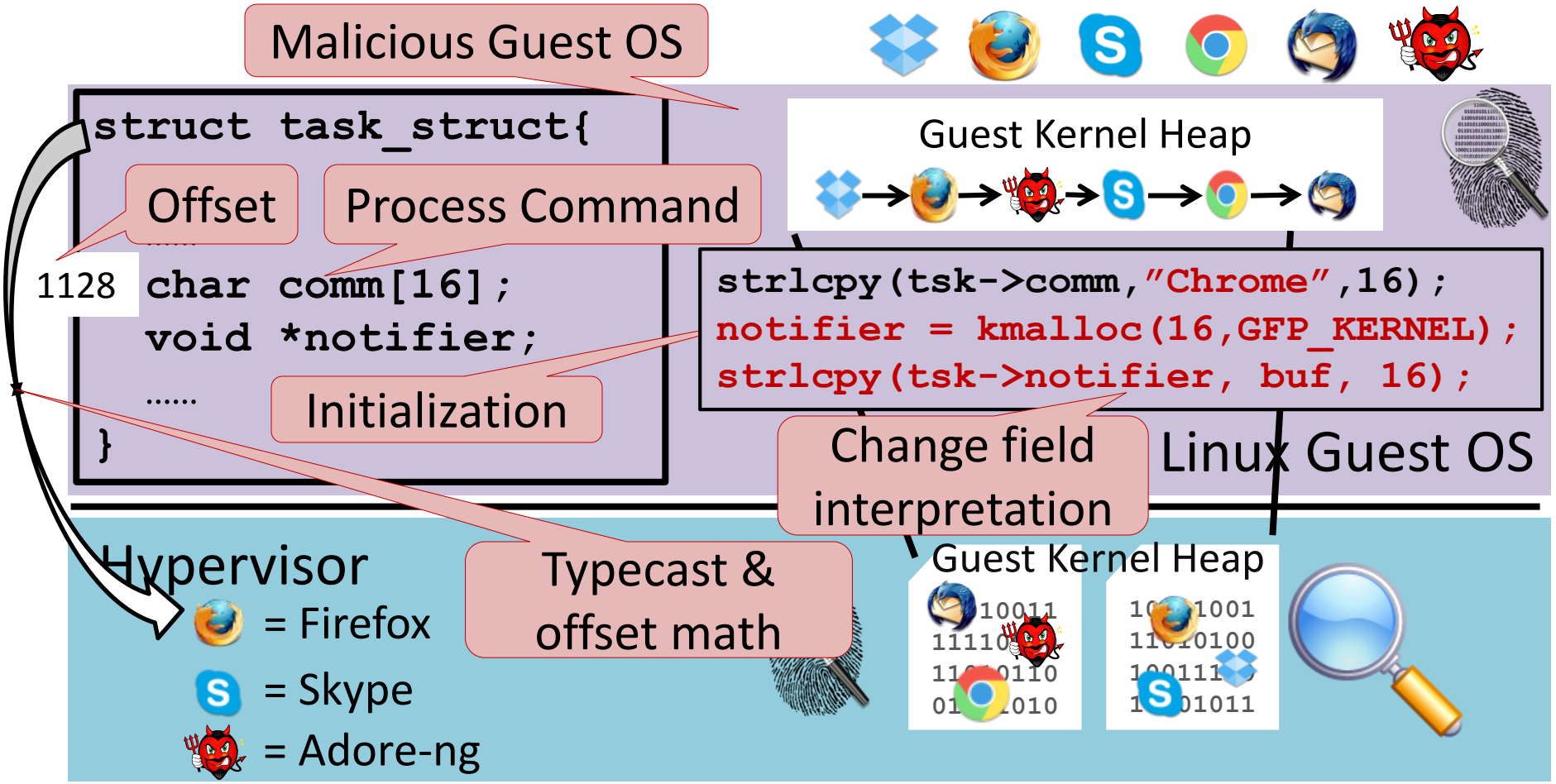
Data Structure Manipulation Attack



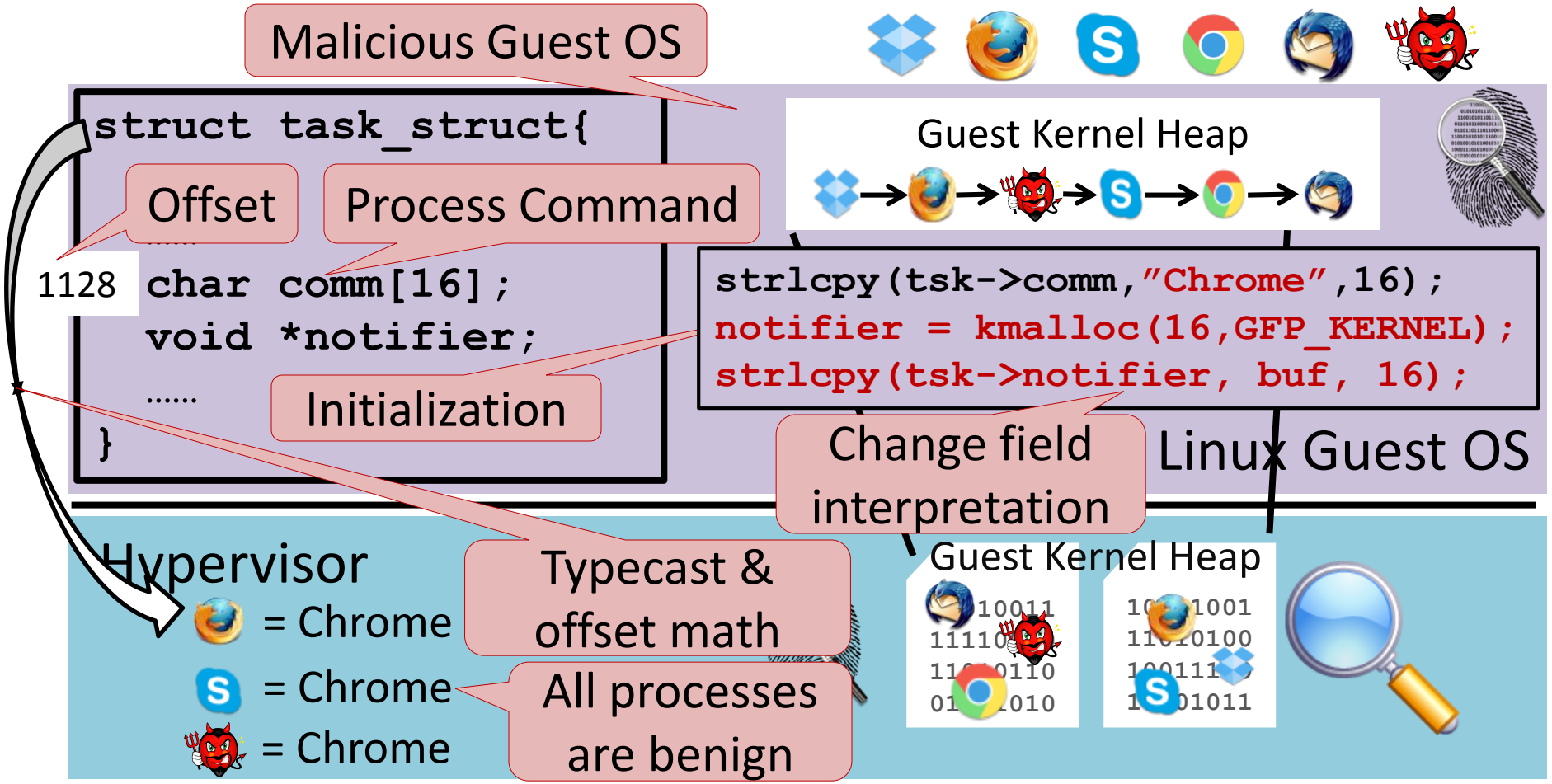
Data Structure Manipulation Attack



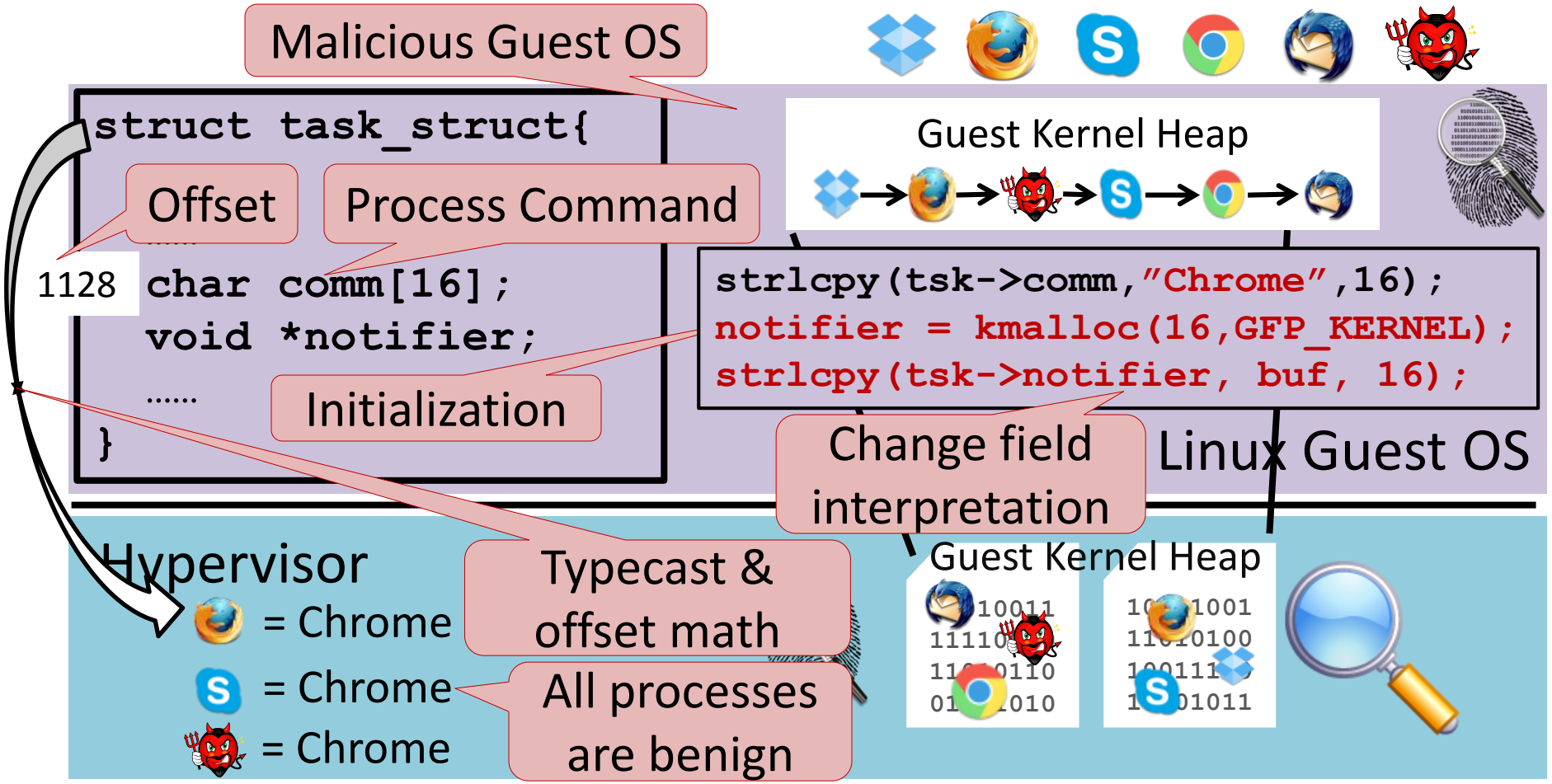
Data Structure Manipulation Attack



Data Structure Manipulation Attack



Data Structure Manipulation Attack



Malicious or Compromised OS can violate VMI assumptions

DKSM Trust Assumptions

- **Assumption:** Consistent structure interpretation
- **Attack**: Change interpretation of a data structure
  - Mislead VMI tools by presenting false system state
- **Defense**: No existing defense
  - CFI on benign kernel may help prevent bootstrapping
  - Attack obviated by generous threat models
  - Trust guest OS to be uncompromised and benign

Structure manipulation: Realistic but outside threat model

Recap

- **Weak Semantic Gap**: **Solved**
  - An engineering challenge
  - Assume guest OS respects data structure templates
- **Strong Semantic Gap**: **Open problem**
  - Malicious or Compromised OS
  - Exploit fragile assumptions to confuse VMI designs

Problem worth working: Strong Semantic Gap

Scalability

- Many VMI designs are fairly expensive
  - Some run sibling VM on dedicated core for analysis
- VMI can be useful in a cloud or multi-VM system
  - Manage overhead & scalability with increase in VMs
- Some VMI systems trade risk to reduce overhead
  - Identify techniques to minimize both overheads & risk

One consideration for VMI systems: Scalabilty

Privacy

- VMI can create new side-channels in cloud
  - Scan period or sibling VM activities using cache timing
- Shouldn't force choice of integrity or privacy risks.
- VMI should evaluate risks of new side channels.
  - Take into account compliance regulation

Another consideration for VMI systems: Privacy

References

[1] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-VM monitoring using hardware virtualization. In CCS, pages 477–487, 2009.

[2] Y. Fu and Z. Lin. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In Oakland, pages 586–600, 2012.

[3] A. Cristina, L. Marziale, G. G. R. Iii, and V. Roussev. Face: Automated digital evidence discovery and correlation. In Digital Forensics, 2005.

[4] The Volatility framework. Online at https://code.google.com/p/volatility/.

[5] W. Cui, M. Peinado, Z. Xu, and E. Chan. Tracking rootkit footprints with a practical memory analysis system. In USENIX Security, pages 42–42, 2012.

[6] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In NDSS, 2011.

[7] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. InCCS, pages 566–577, 2009

[8] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In SOSP, pages 335–350, 2007

[9] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In CCS, pages 545–554, 2009.

[10] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with OSck. In ASPLOS, pages 279–290, 2011