Kernel Malware Analysis with Un-tampered and Temporal Views of Dynamic Kernel Memory

[+]Junghwan Rhee, [*]Ryan Riley, [+]Dongyan Xu, [**]Xuxian Jiang

[+] Department of Computer Science, Purdue University

[*] Department of Computer Science and Engineering, Qatar University

[**] Department of Computer Science, North Carolina State University

Outline

- Background
- Allocation-driven mapping
- Evaluation
- Discussion
- Conclusion
- Demo

Kernel malware

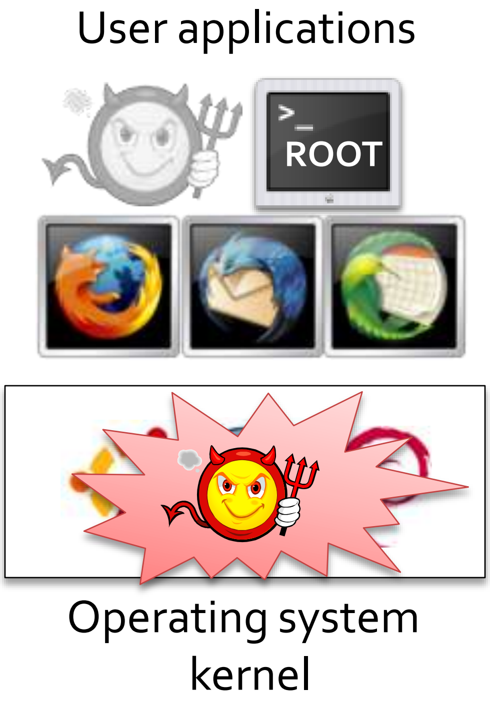
- Kernel malware attacks operating system kernels.
  - e.g., kernel rootkits
- Attack goals
  - Hide processes, files, etc.
  - Provide hidden services, backdoors, etc.
- Attack techniques
  - Hijack system services (e.g., system calls)
  - Directly manipulate kernel data (DKOM)
  - Hijack hooks by overwriting function pointers (KOH)

User applications

Operating system kernel

Kernel malware

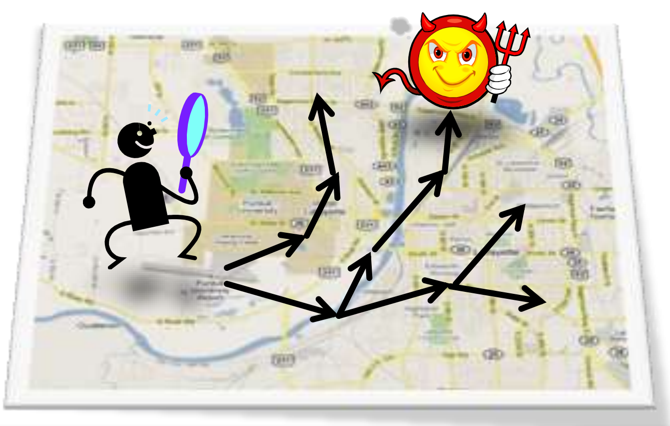
- Kernel malware attacks operating system kernels.
  - e.g., kernel rootkits
- Attack goals
  - Hide processes, files, etc.
  - Provide hidden services, backdoors, etc.
- Attack techniques
  - Hijack system services (e.g., system calls)
  - Directly manipulate kernel data (DKOM)
  - Hijack hooks by overwriting function pointers (KOH)



Kernel memory mapping

- Kernel memory mapping has been used for kernel integrity checking and kernel malware detection.

- Existing approaches
  - **Type-projection mapping**: kernel objects identification by recursively traversing pointers from global objects
    - Static: memory snapshots as input
    - Dynamic: memory traces as input

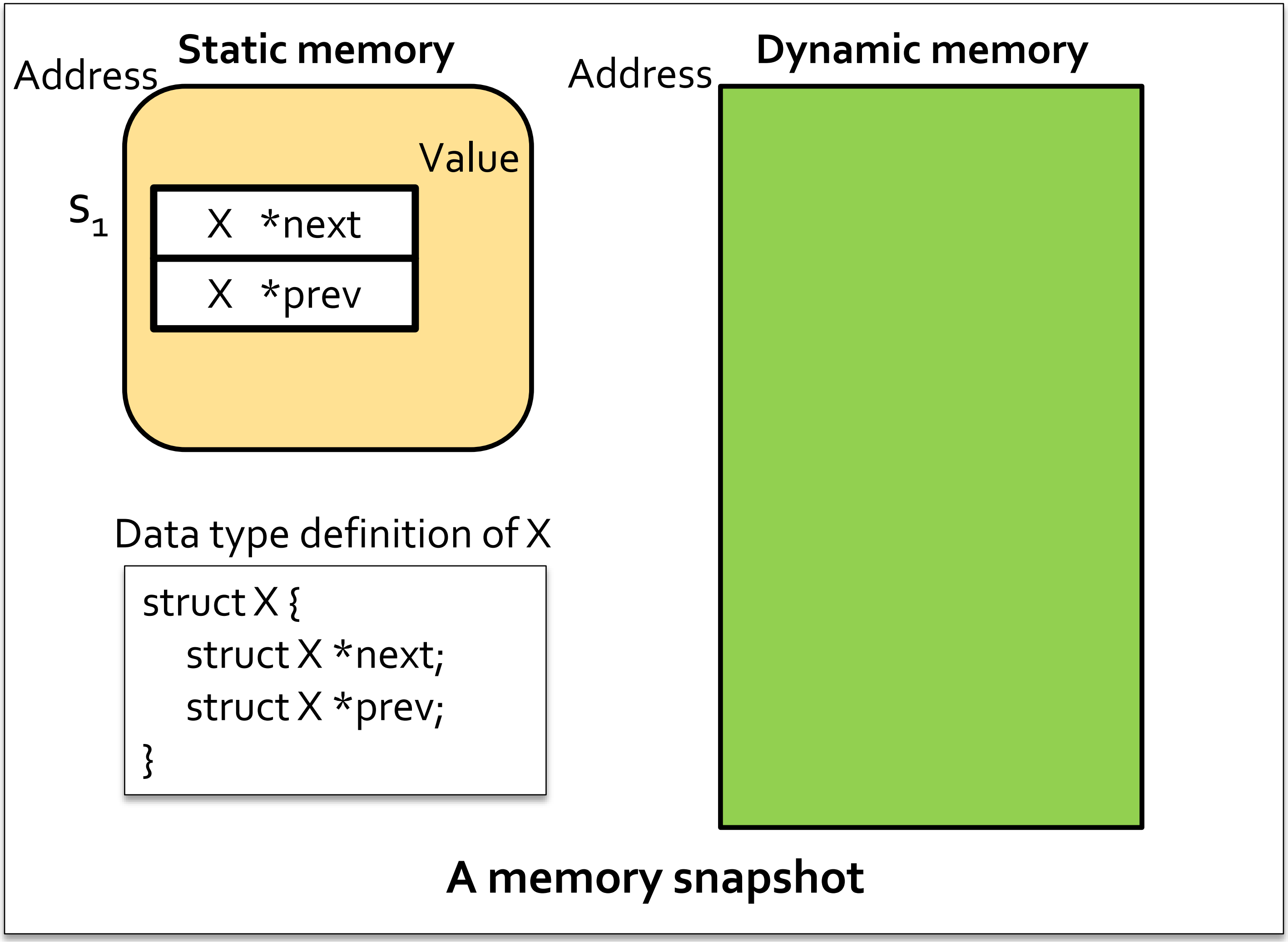


Related work

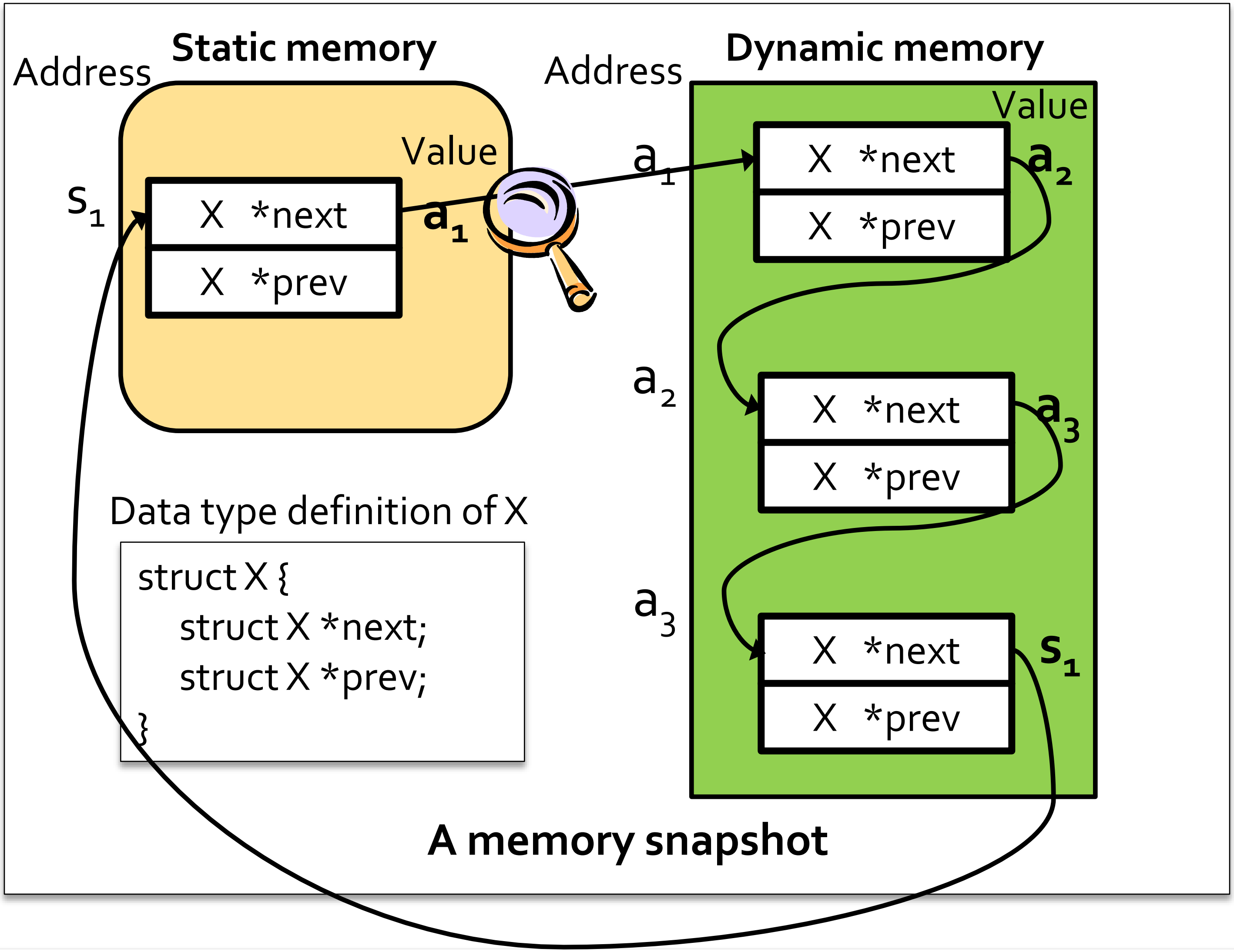
- Type-projection mapping using memory snapshots
  - SBCFI [CCS 2007]
  - Gibraltar [ACSAC 2008]
  - KOP [CCS 2009]

- Type-projection mapping using memory traces
  - Rkprofiler [RAID 2009]
  - PoKeR [Eurosys 2009]

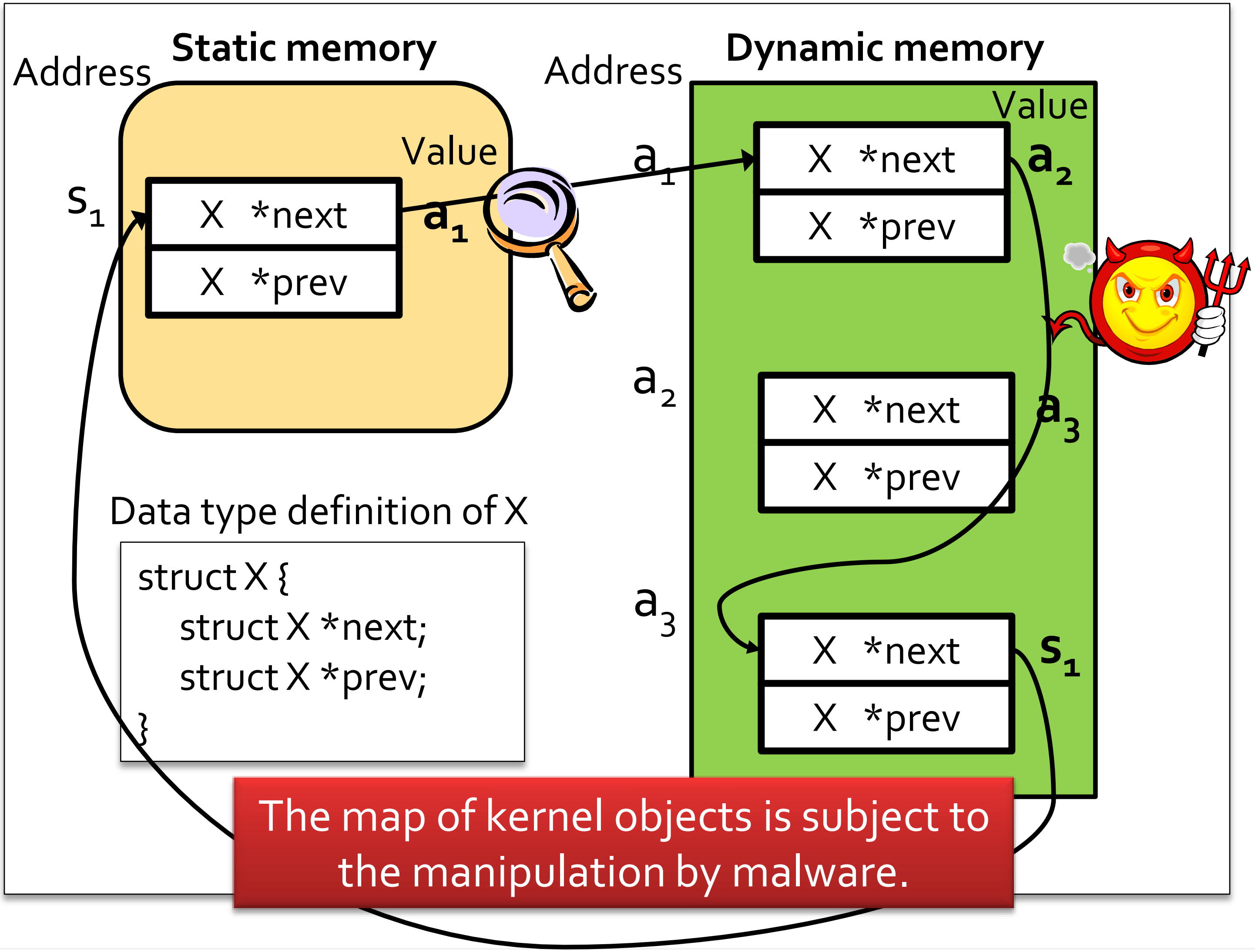
Type-projection mapping



Type-projection mapping

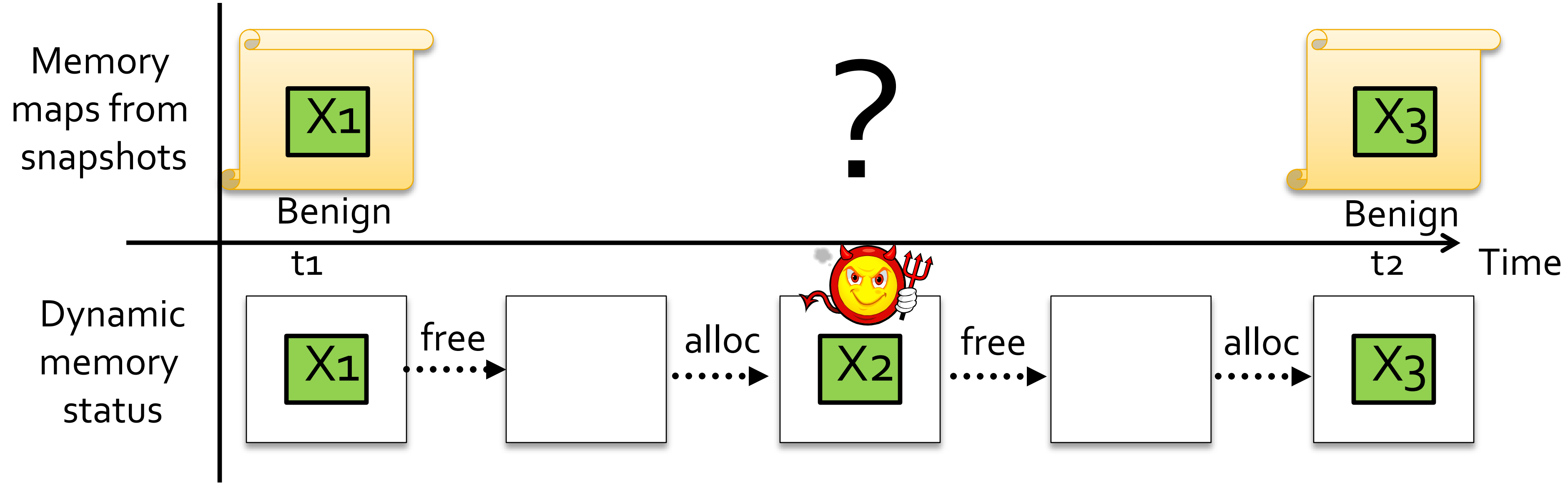


Challenge : Memory manipulation



The map of kernel objects is subject to the manipulation by malware.

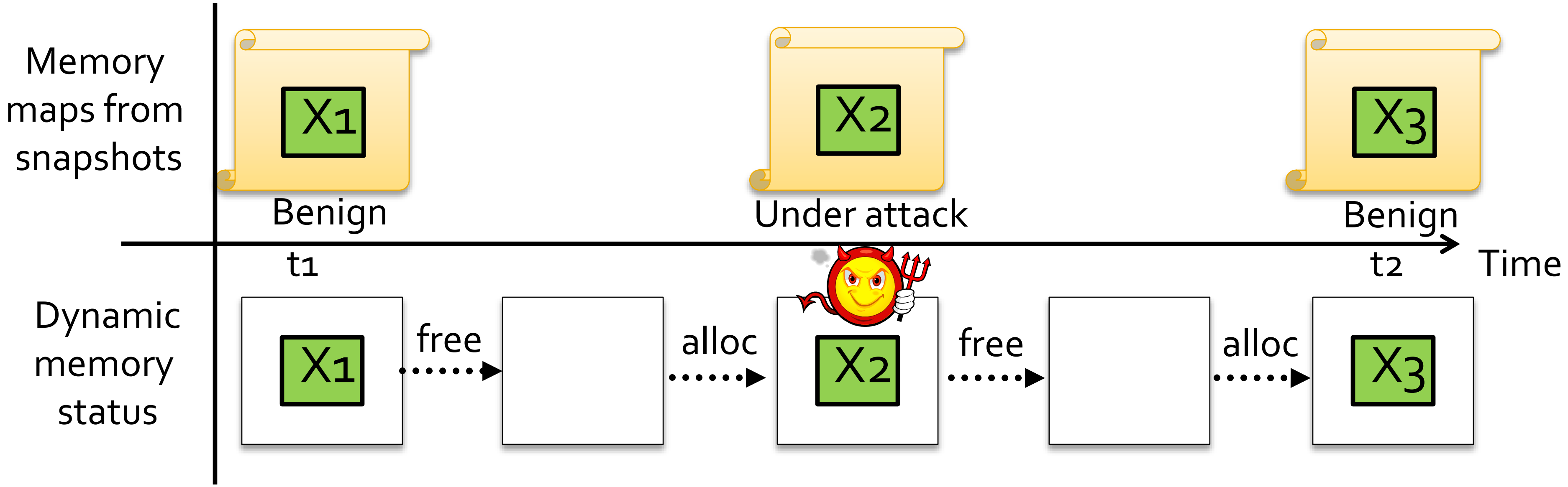
Challenge : Asynchronous mapping



Malware analysis using an asynchronous mapping

- X1, X2, and X3 : kernel objects allocated in the ***same address*** with the ***same data type***.
- A malware analyzer based on asynchronous mapping may not be able to differentiate X1, X2, and X3.

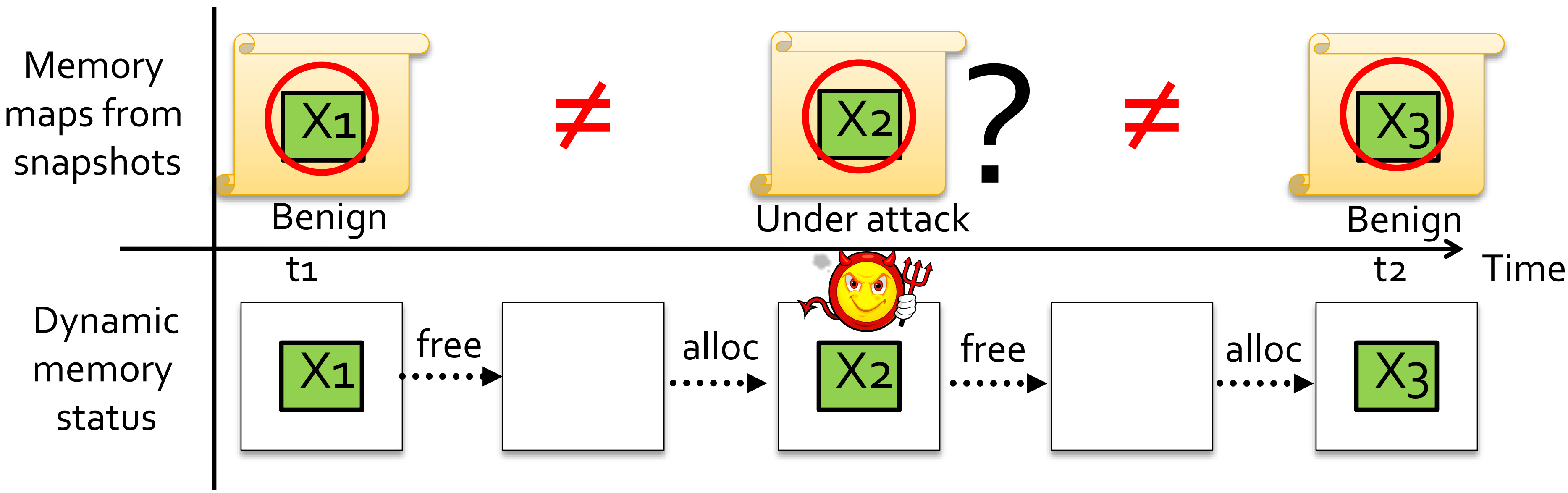
Challenge : Asynchronous mapping



Malware analysis using an asynchronous mapping

- X1, X2, and X3 : kernel objects allocated in the ***same address*** with the ***same data type***.
- A malware analyzer based on asynchronous mapping may not be able to differentiate X1, X2, and X3.

Challenge : Asynchronous mapping



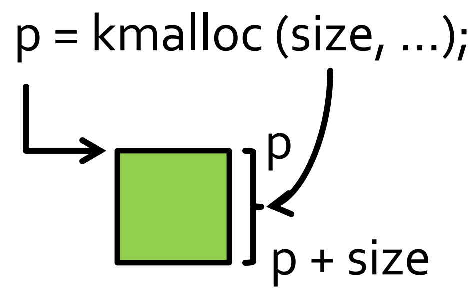
Malware analysis using an asynchronous mapping

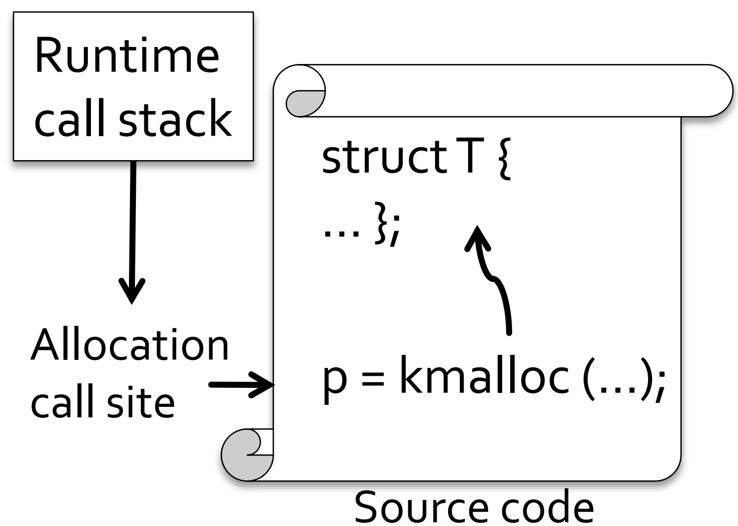
- X1, X2, and X3 : kernel objects allocated in the ***same address*** with the ***same data type***.
- A malware analyzer based on asynchronous mapping may not be able to differentiate X1, X2, and X3.

Our solution: Allocation-driven mapping

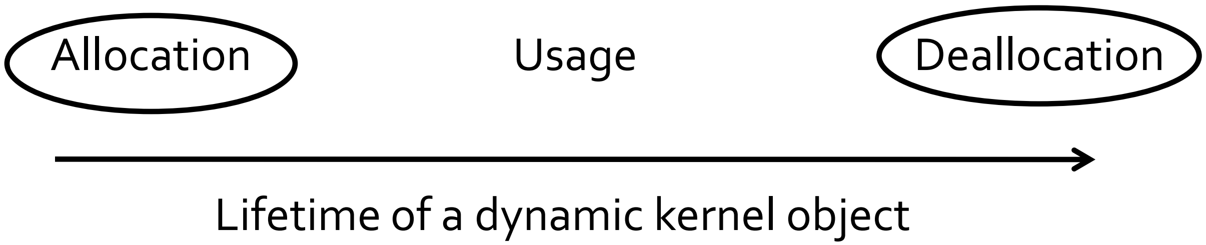
- Kernel objects are identified by transparently capturing kernel memory function calls.
- The memory ranges are extracted from function arguments and return values.
- Call stack information (allocation call site) is used to derive data types.

* An memory allocation call site:
code address of a memory allocation call

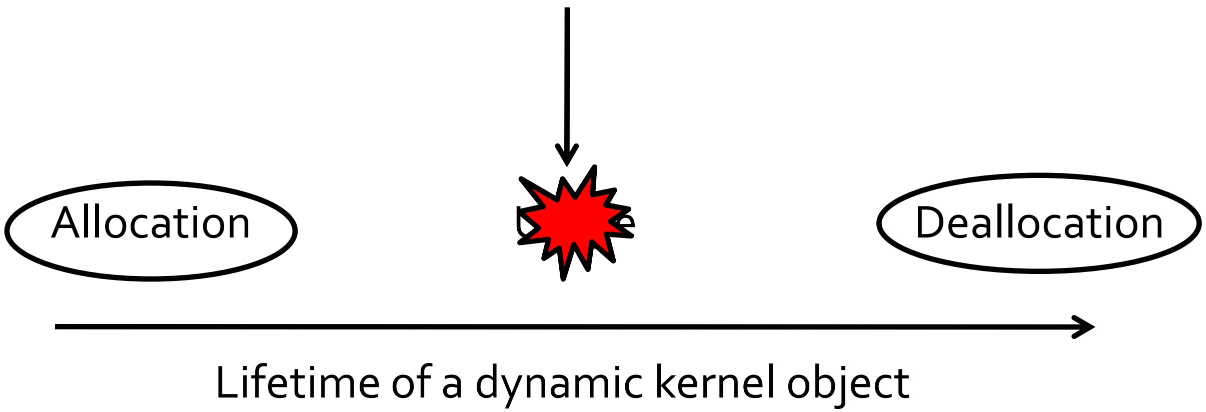




Allocation-driven mapping

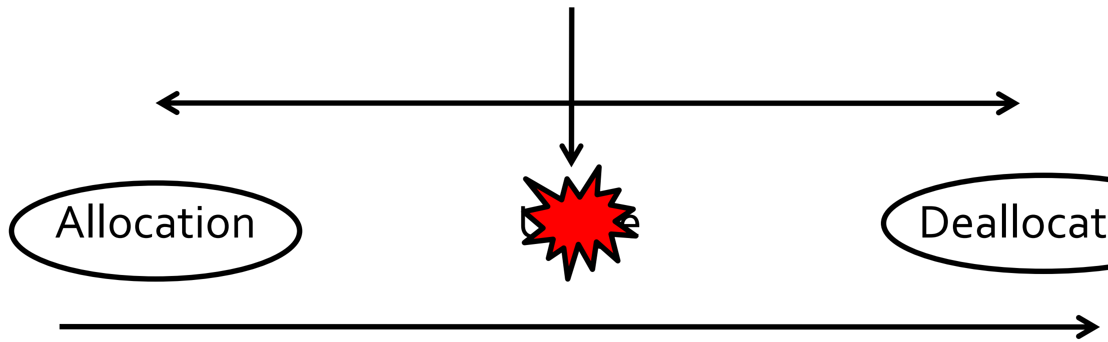


Allocation-driven mapping



- Advantages
  - Un-tampered view
    - Tolerant to the manipulation of memory content

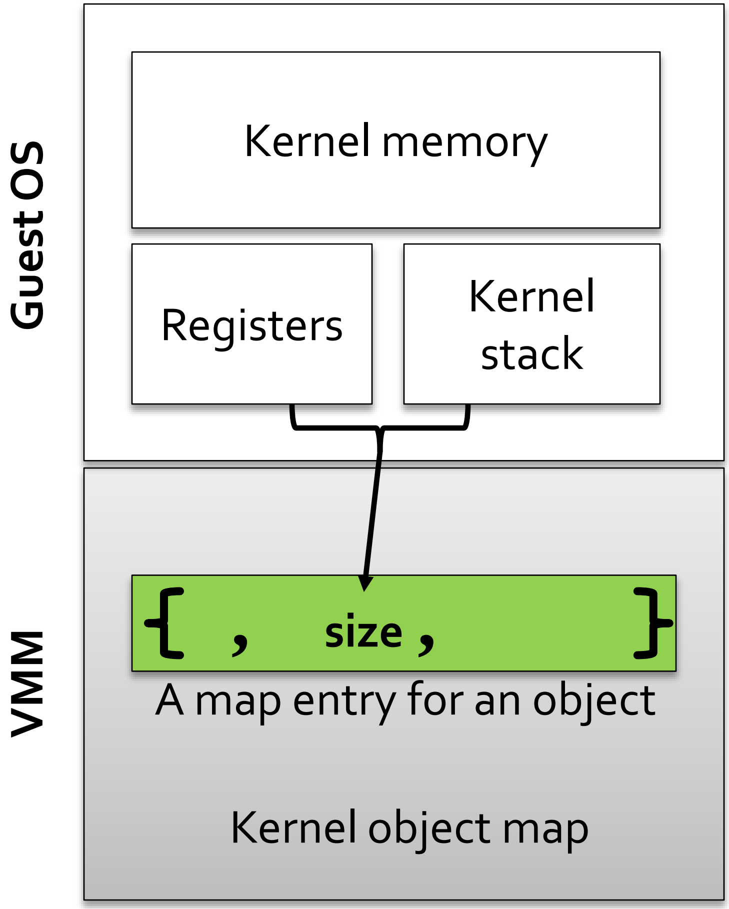
Allocation-driven mapping



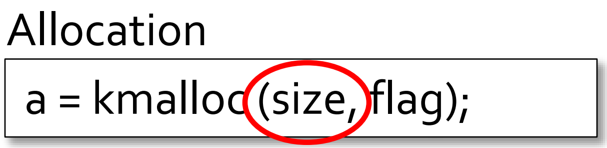
Lifetime of a dynamic kernel object

- Advantages
  - Un-tampered view
    - Tolerant to the manipulation of memory content
  - Temporal view
    - Lifetime of dynamic data is tracked to differentiate objects at the same memory location

Techniques : Map generation

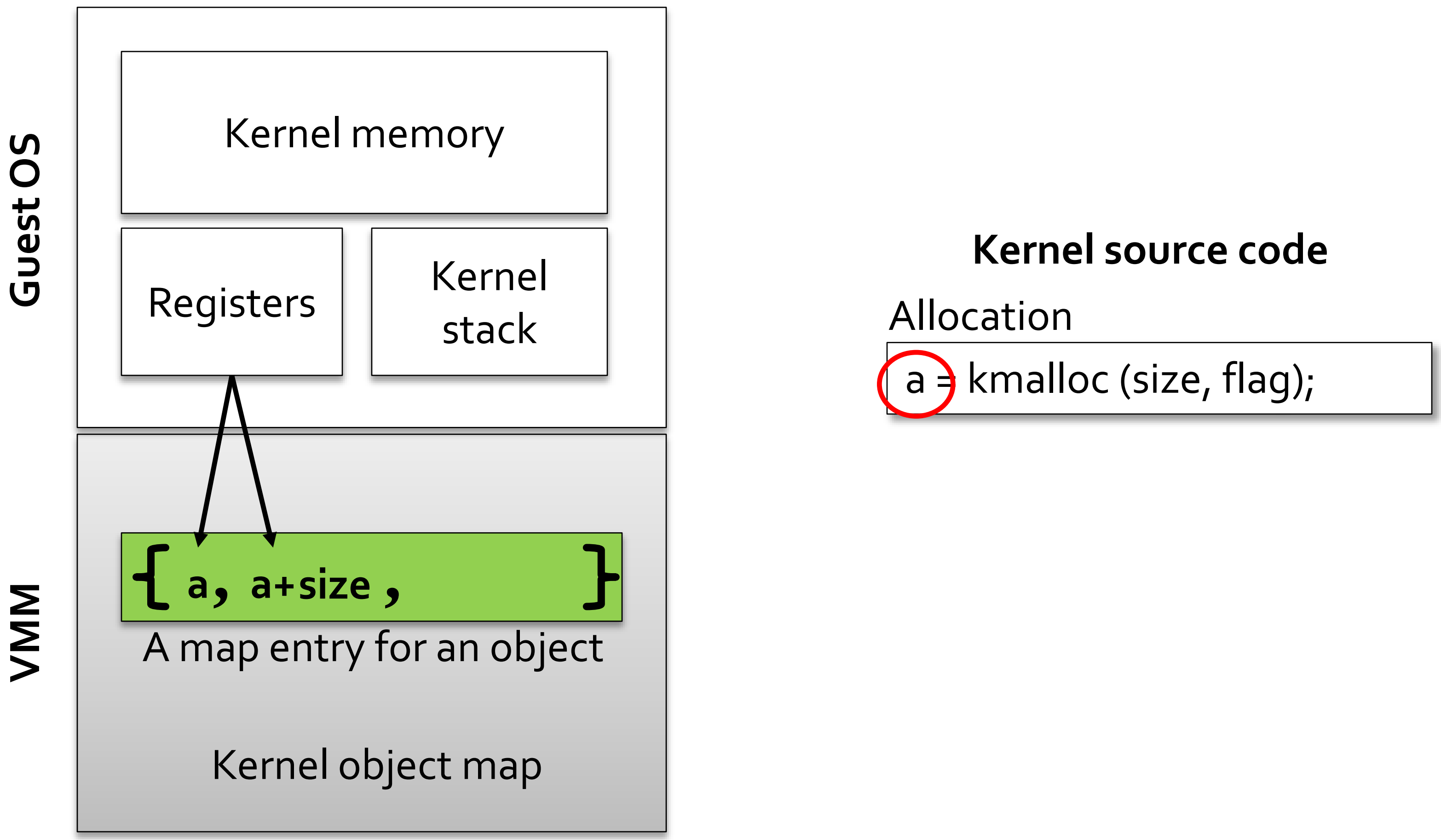


Kernel source code



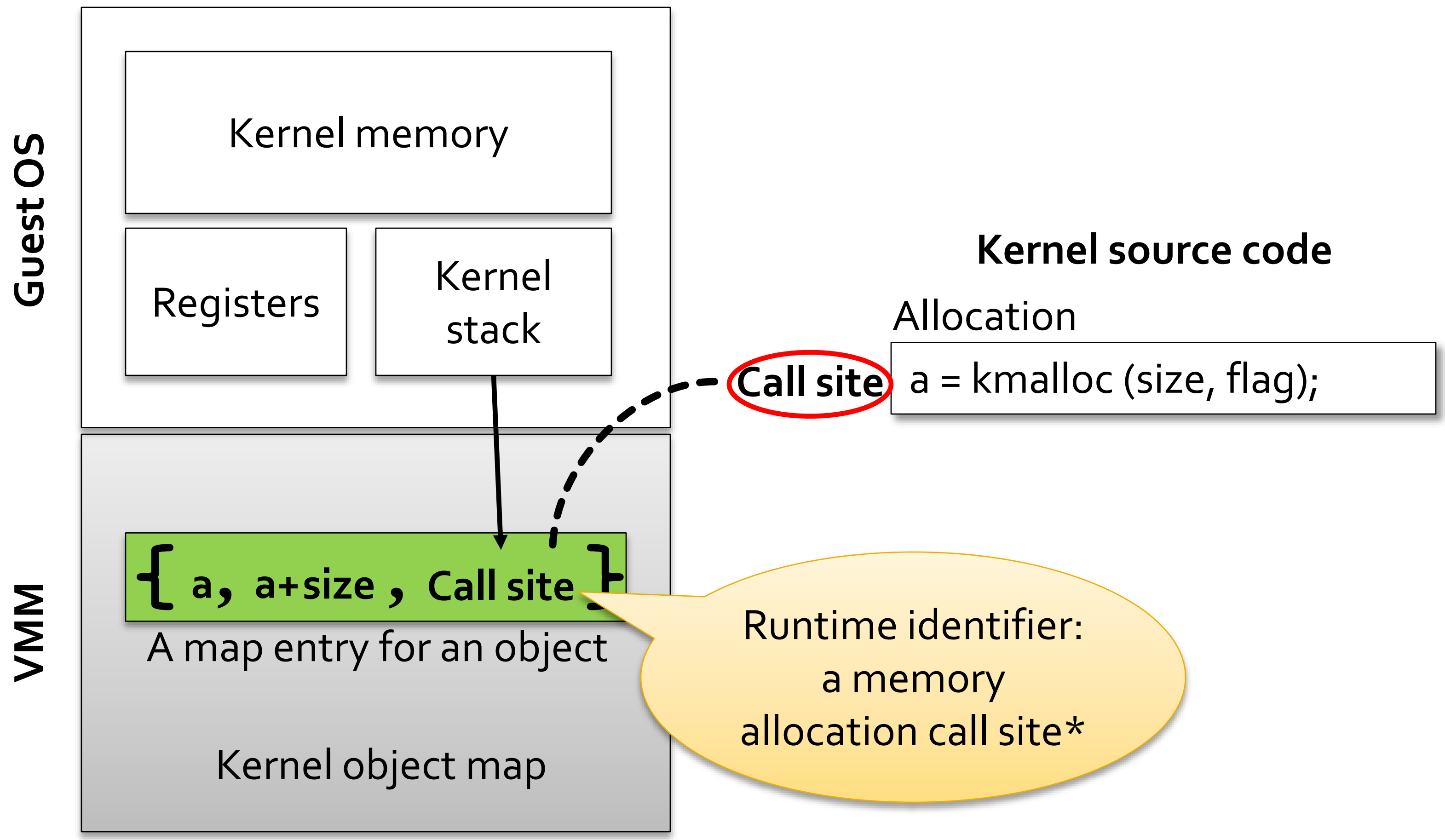
* An memory allocation call site: code address of a memory allocation call

Techniques : Map generation



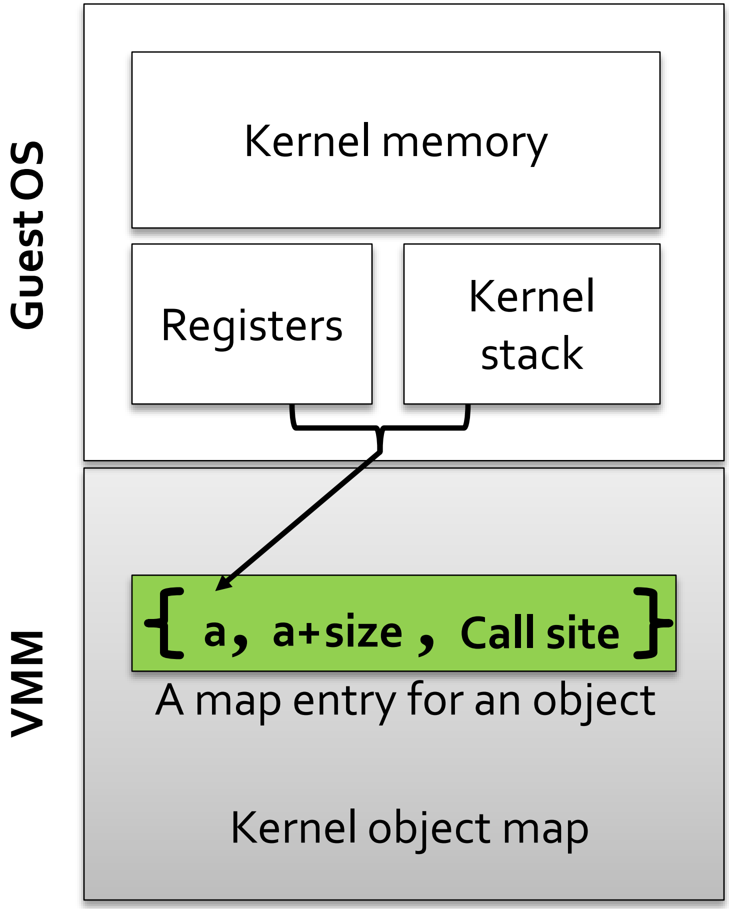
* An memory allocation call site: code address of a memory allocation call

Techniques : Map generation



* An memory allocation call site: code address of a memory allocation call

Techniques : Map generation



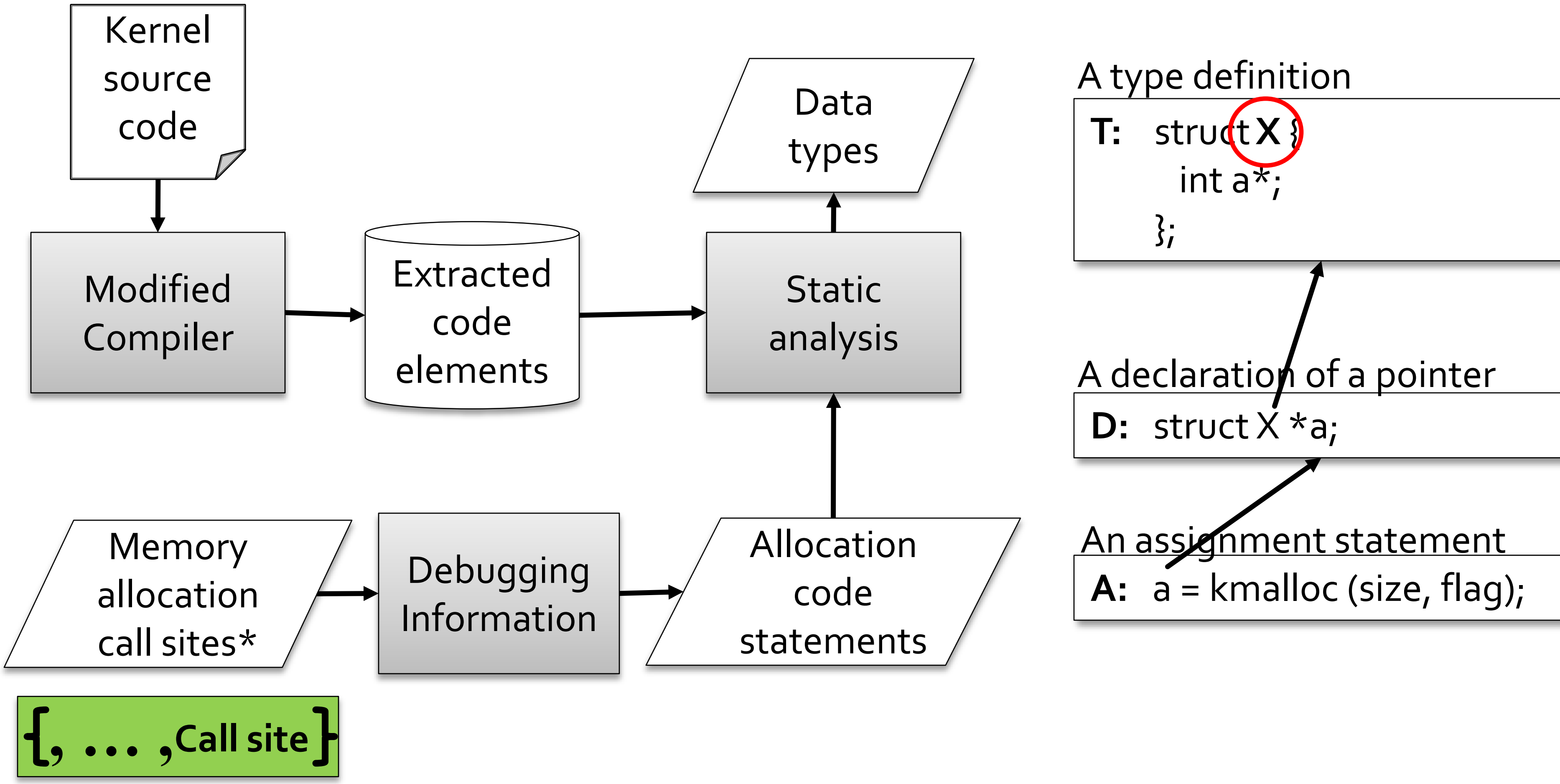
Kernel source code

Deallocation

kfree(a);

* An memory allocation call site: code address of a memory allocation call

Techniques : Type derivation



* An memory allocation call site: code address of a memory allocation call

Implementation

- LiveDM : Live Dynamic kernel memory Map
- Supported guest OS kernels
  - Redhat 8, Debian Sarge, Fedora Core 6
- Virtual machine monitor : QEMU
- Knowledge of kernel memory functions is assumed.
- Type resolution
  - Debugging symbols for translation of allocation call sites
  - Modified gcc compiler to extract code elements

Evaluation

- Effectiveness
- Performance
- Applications
  - Hidden object detector (un-tampered view)
  - Temporal malware behavior monitor (temporal view)

Evaluation : Identifying objects

Type resolution

A → D → T

Identified instances

`kernel/fork.c:248`

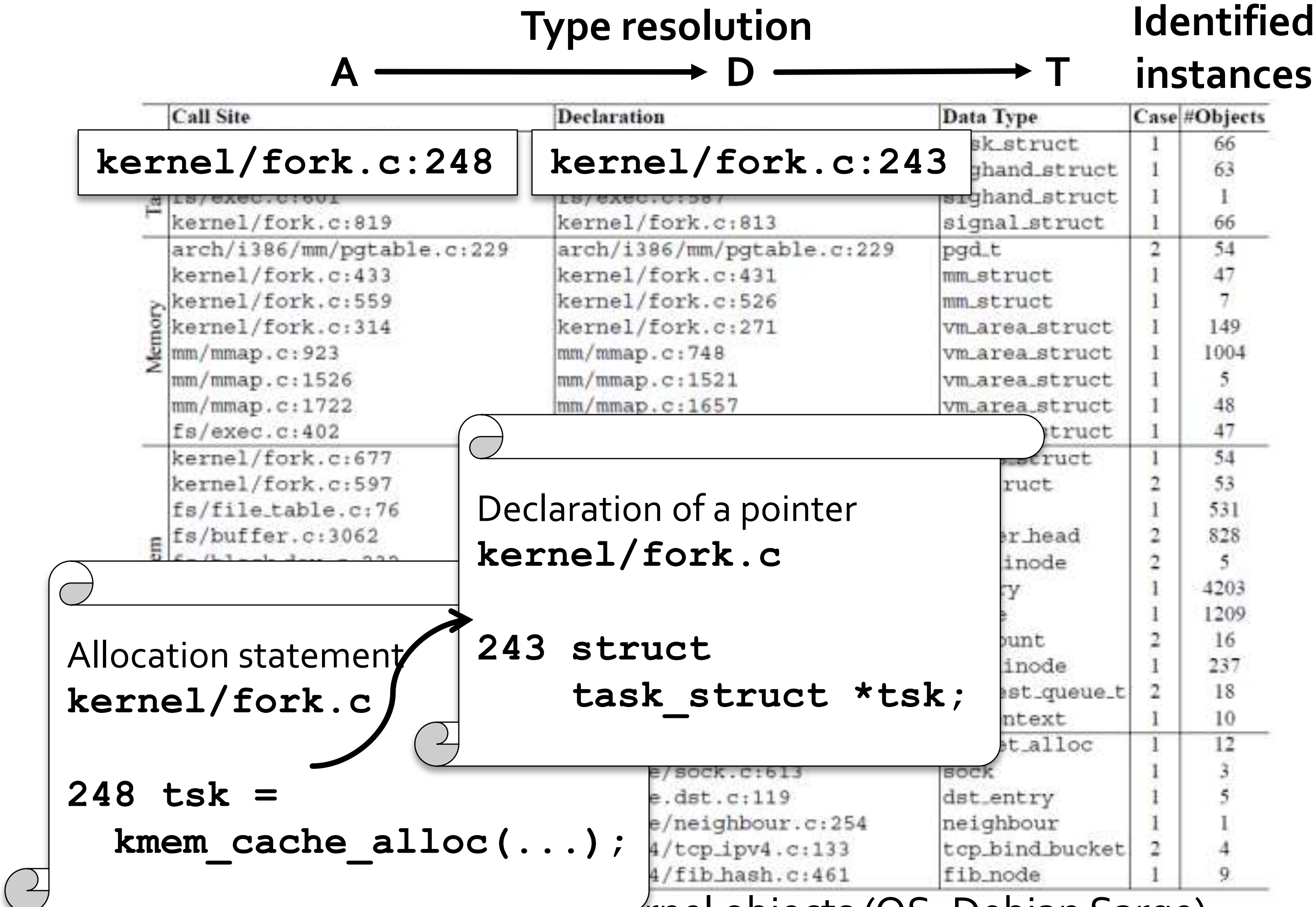
	Call Site	Declaration	Data Type	Case	#Objects
Ta...	[illegible]	kernel/fork.c:243	task_struct	1	66
	[illegible]	kernel/fork.c:795	sighand_struct	1	63
	fs/exec.c:601	fs/exec.c:587	sighand_struct	1	1
	kernel/fork.c:819	kernel/fork.c:813	signal_struct	1	66
Memory	arch/i386/mm/pgtable.c:229	arch/i386/mm/pgtable.c:229	pgd_t	2	54
	kernel/fork.c:433	kernel/fork.c:431	mm_struct	1	47
	kernel/fork.c:559	kernel/fork.c:526	mm_struct	1	7
	kernel/fork.c:314	kernel/fork.c:271	vm_area_struct	1	149
	mm/mmap.c:923	mm/mmap.c:748	vm_area_struct	1	1004
	mm/mmap.c:1526	mm/mmap.c:1521	vm_area_struct	1	5
	mm/mmap.c:1722	mm/mmap.c:1657	vm_area_struct	1	48
	fs/exec.c:402	fs/exec.c:342	vm_area_struct	1	47
...em	kernel/fork.c:677	kernel/fork.c:654	files_struct	1	54
	kernel/fork.c:597	kernel/fork.c:597	fs_struct	2	53
	fs/file_table.c:76	fs/file_table.c:69	file	1	531
	fs/buffer.c:3062	fs/buffer.c:3062	buffer_head	2	828
	[illegible]	...ev.c:232	bdev_inode	2	5
	[illegible]	...:689	dentry	1	4203
	[illegible]	...e.c:107	inode	1	1209
	[illegible]	...space.c:55	vfsmount	2	16
	[illegible]	.../inode.c:90	proc_inode	1	237
	[illegible]	.../block/ll_rw_blk.c:1405	request_queue_t	2	18
	[illegible]	.../block/ll_rw_blk.c:2945	io_context	1	10
	[illegible]	...ket.c:278	socket_alloc	1	12
	[illegible]	...e/sock.c:613	sock	1	3
	[illegible]	...e.dst.c:119	dst_entry	1	5
	[illegible]	...e/neighbour.c:254	neighbour	1	1
	[illegible]	...4/tcp_ipv4.c:133	tcp_bind_bucket	2	4
	[illegible]	...4/fib_hash.c:461	fib_node	1	9

Allocation statement
kernel/fork.c

```
248 tsk =
  kmem_cache_alloc(...);
```

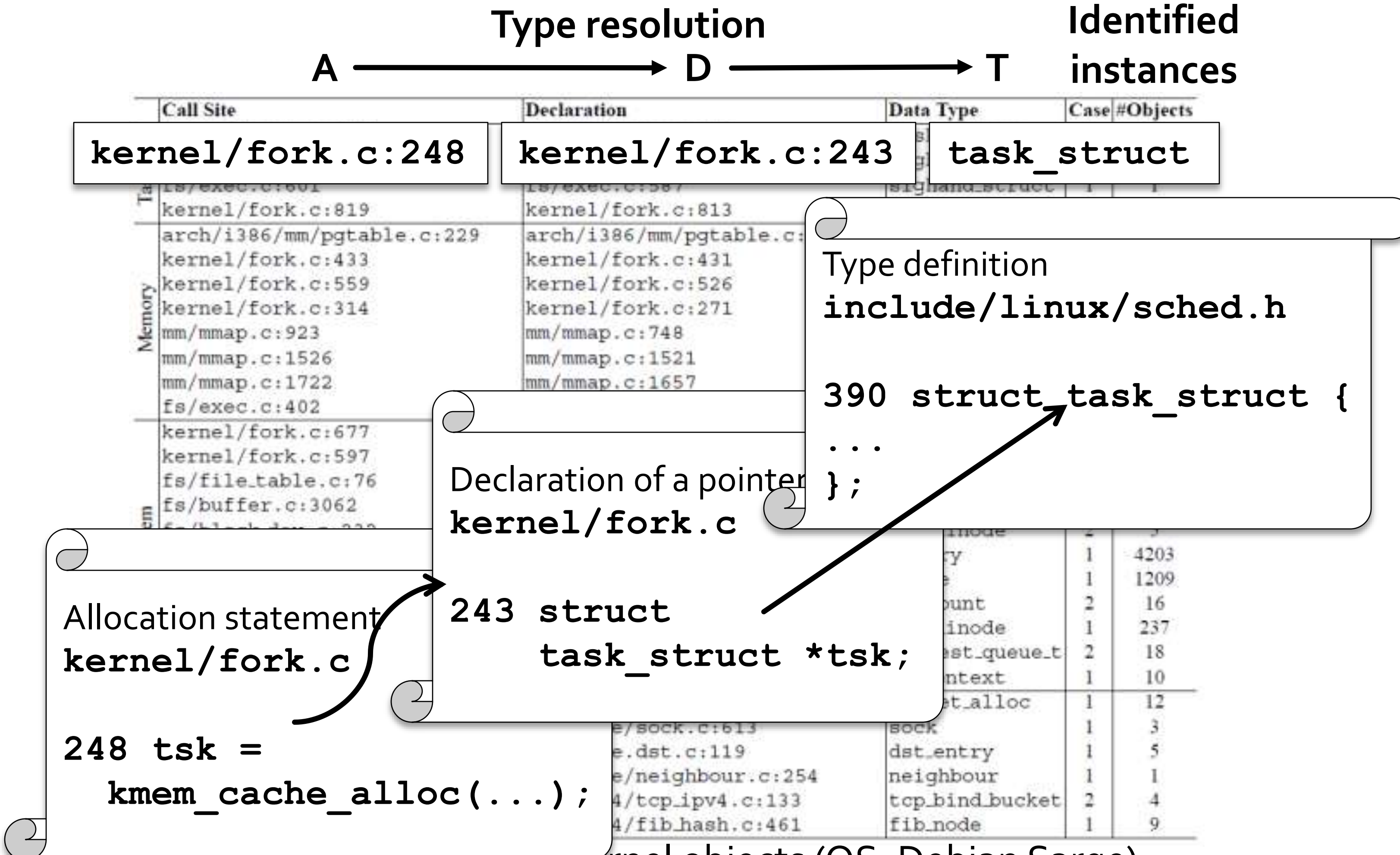
A list of core dynamic kernel objects (OS: Debian Sarge)
Total dynamic kernel objects: 29488

Evaluation : Identifying objects



A list of core dynamic kernel objects (OS: Debian Sarge)
Total dynamic kernel objects: 29488

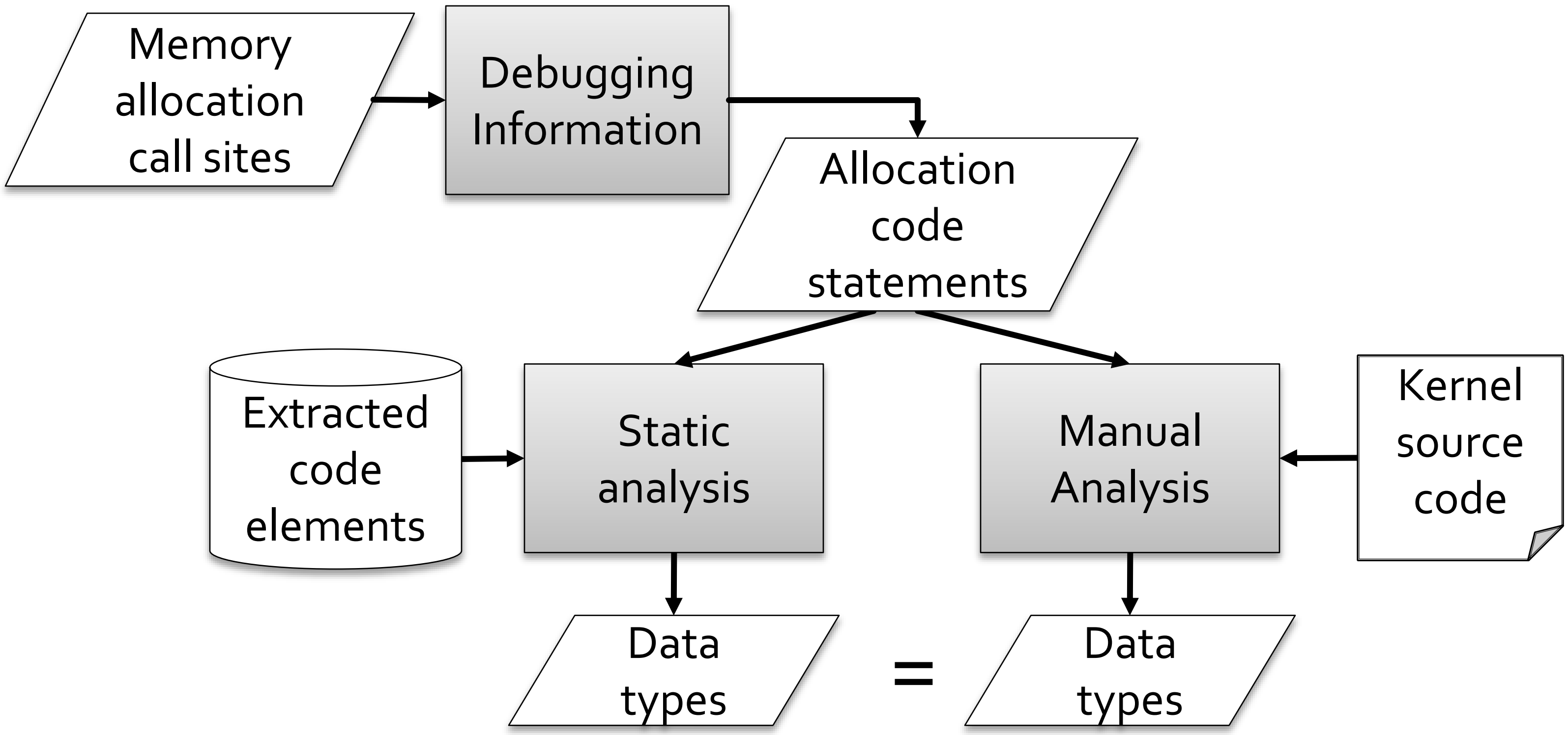
Evaluation : Identifying objects



A list of core dynamic kernel objects (OS: Debian Sarge)

Total dynamic kernel objects: 29488

Evaluation : Type resolution



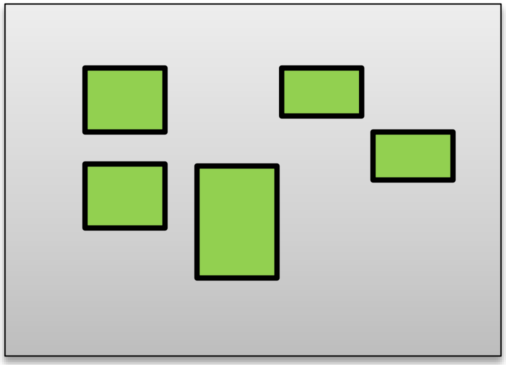
- Manual analysis: convert allocation call sites to data types (similar to validation methods of KOP [Carbone et. al., CCS 2009] and Laika [Cozzie et. al., OSDI 2008])

Evaluation : Performance

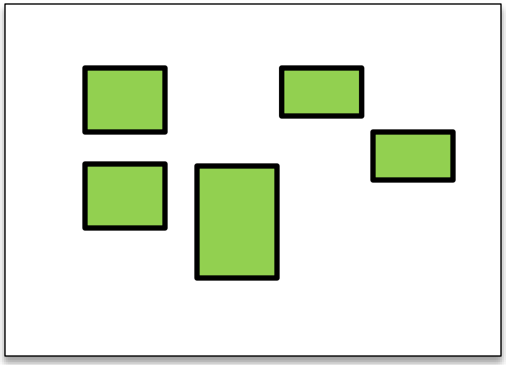
- Benchmarks
  - Kernel compile, UnixBench, nbench
- Overhead
  - Slowdown compared to unmodified QEMU (worst in benchmarks): 42% for Linux 2.4, 125% for Linux 2.6
  - Mainly caused by the capture of dynamic objects
  - Near-zero overhead for CPU-intensive benchmarks
- Non-production application scenarios
  - Honeypot, malware profiling, kernel debugging

An application of the un-tampered view

Allocation-driven map

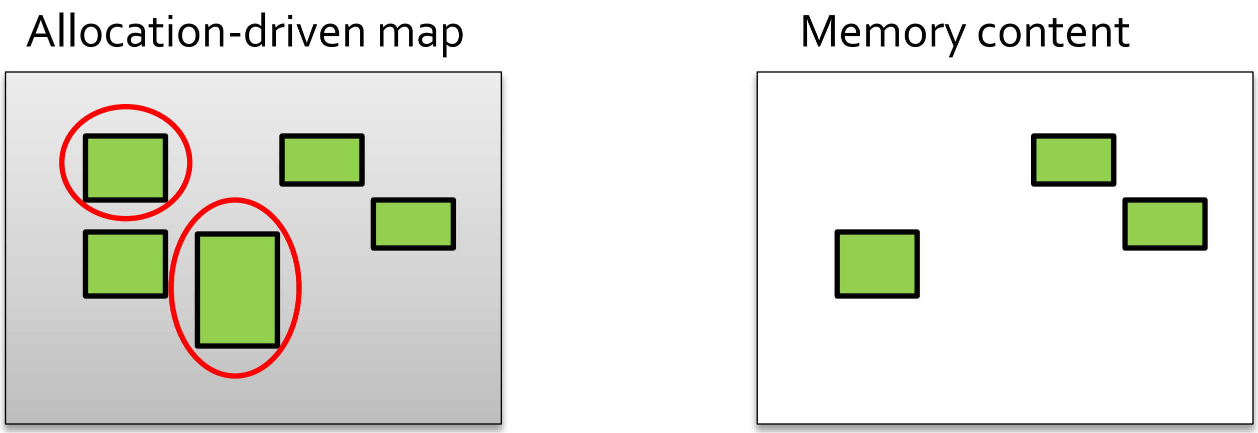


Memory content



- Hidden object detector
  - Periodic comparison of an allocation-driven map and memory content

An application of the un-tampered view



- Hidden object detector
  - Periodic comparison of an allocation-driven map and memory content

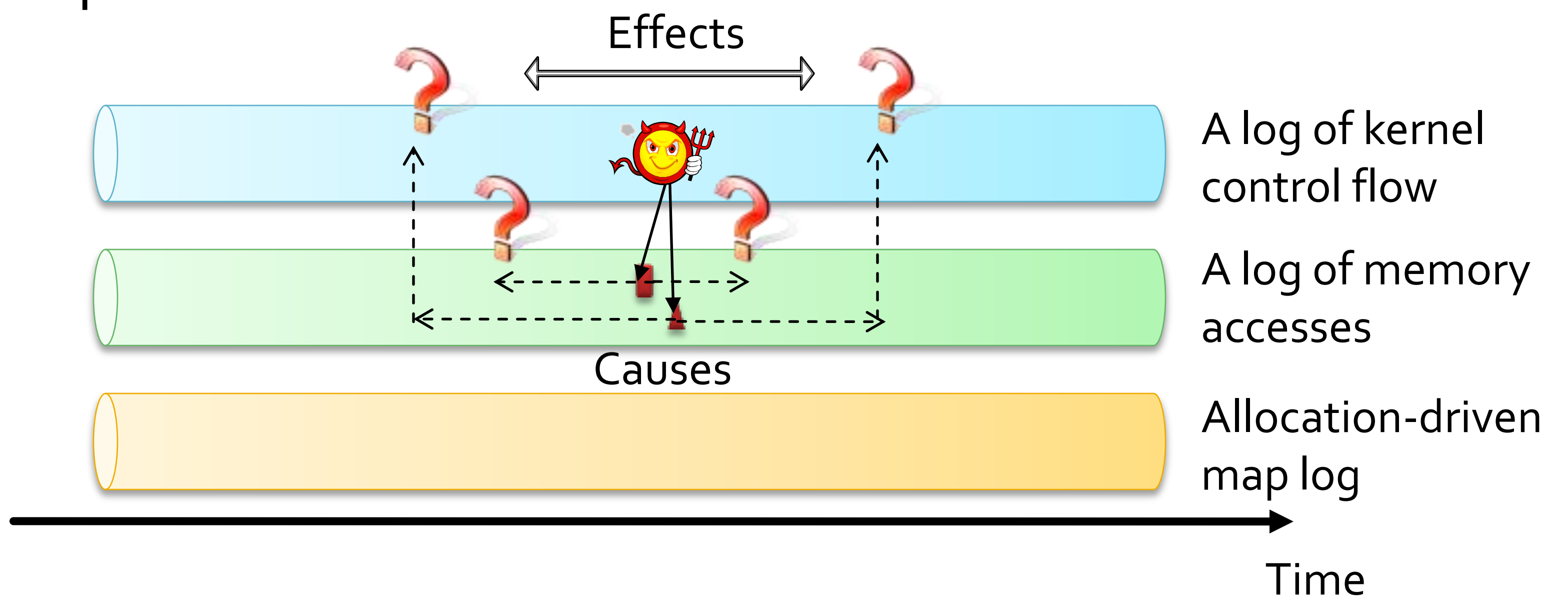
An application of the un-tampered view

Rootkit Name	|L| - |S|	Manipulated Data		Operating System	Attack Vector
		Type	Field		
hide_lkm	# of hidden modules	module	next	Redhat 8	/dev/kmem
fuuld	# of hidden PCBs	task_struct	next_task, prev_task	Redhat 8	/dev/kmem
cleaner	# of hidden modules	module	next	Redhat 8	LKM
modhide	# of hidden modules	module	next	Redhat 8	LKM
hp 1.0.0	# of hidden PCBs	task_struct	next_task, prev_task	Redhat 8	LKM
linuxfu	# of hidden PCBs	task_struct	next_task, prev_task	Redhat 8	LKM
modhide1	1 (rootkit self-hiding)	module	next	Redhat 8	LKM
kis 0.9 (server)	1 (rootkit self-hiding)	module	next	Redhat 8	LKM
adore-ng-2.6	1 (rootkit self-hiding)	module	list.next, list.prev	Debian Sarge	LKM
ENYELKM 1.1	1 (rootkit self-hiding)	module	list.next, list.prev	Debian Sarge	LKM

- Hidden object detector
  - Periodic comparison of an allocation-driven map and memory content
  - 10 kernel rootkits are tested and all detected.
  - Agnostic to the injection of malware code
  - Non-code injection attacks (hide_lkm and fuuld) are detected.

An application of the temporal view

- Temporal Malware Behavior Monitor
  - Systematically visualize malware influence *via the manipulation of dynamic kernel memory*
  - Steps



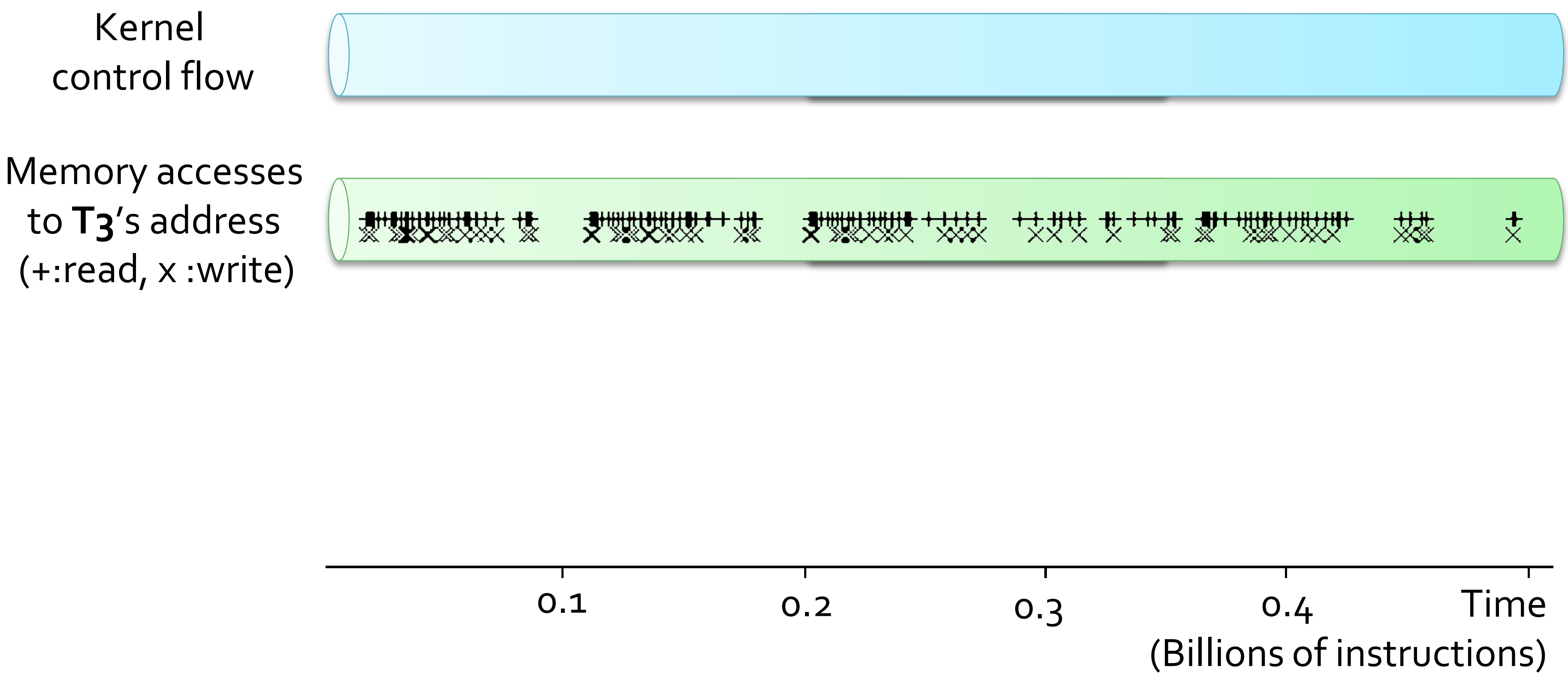
An application of the temporal view

- Temporal Malware Behavior Monitor
  - Systematically visualize malware influence *via the manipulation of dynamic kernel memory*

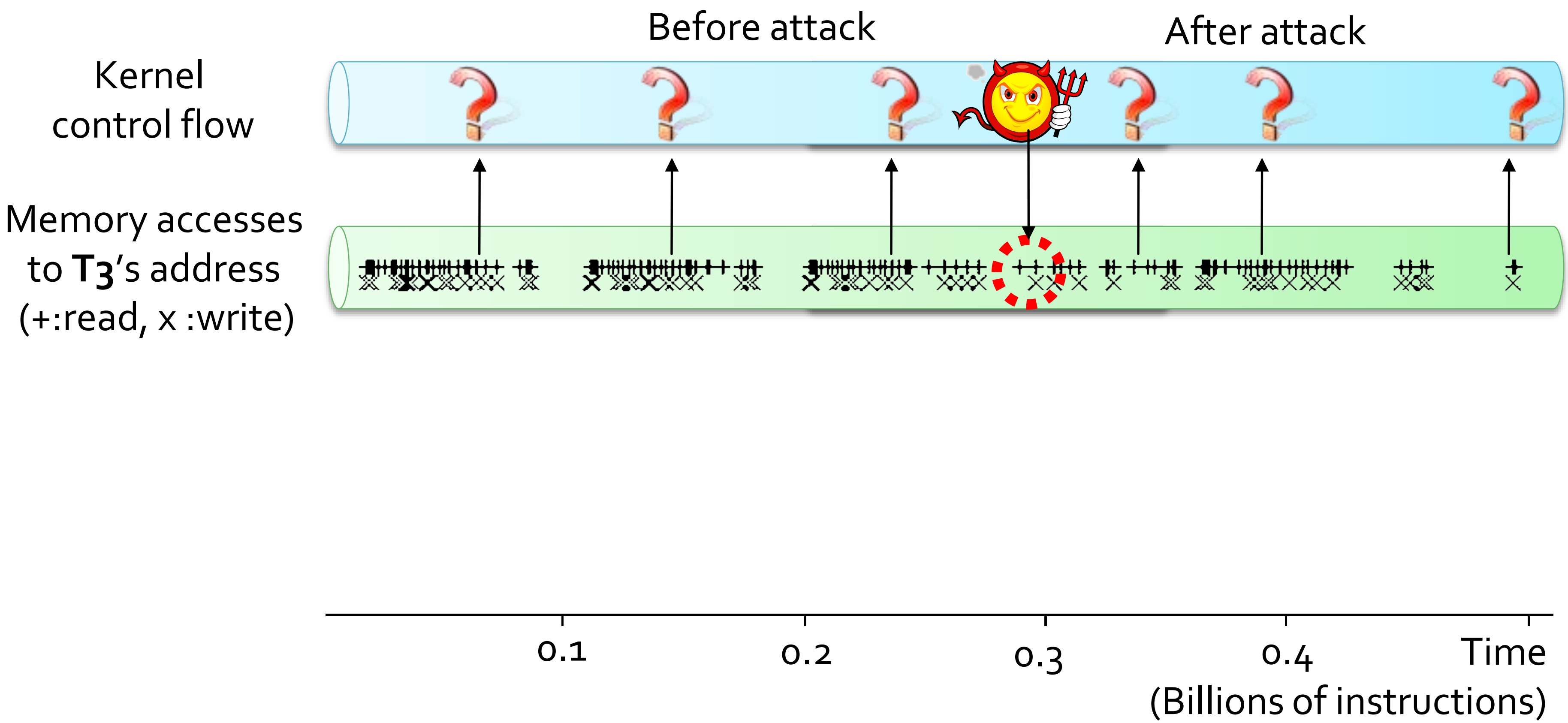
Runtime Identification		Offline Data Type Interpretation		
Call Site	**Offset**	**Type / Object (Static, Module object)**	**Field**	
fork.c:610	0x4,12c,130	task_struct	flags,uid,euid	
fork.c:610	0x134,138,13c	task_struct	suid,fsuid,gid	
fork.c:610	0x140,144,148	task_struct	egid,sgid,fsgid	
fork.c:610	0x1d0	task_struct	cap_effective	T3
fork.c:610	0x1d4	task_struct	cap_inheritable	
fork.c:610	0x1d8	task_struct	cap_permitted	
generic.c:436	0x20	proc_dir_entry	get_info	
(Static object)		proc_root_inode_operations	lookup	
(Static object)		proc_root_operations	readdir	
(Static object)		unix_dgram_ops	recvmsg	
(Module object)		ext3_dir_operations	readdir	
(Module object)		ext3_file_operations	write	

The list of kernel objects manipulated by adore-ng rootkit

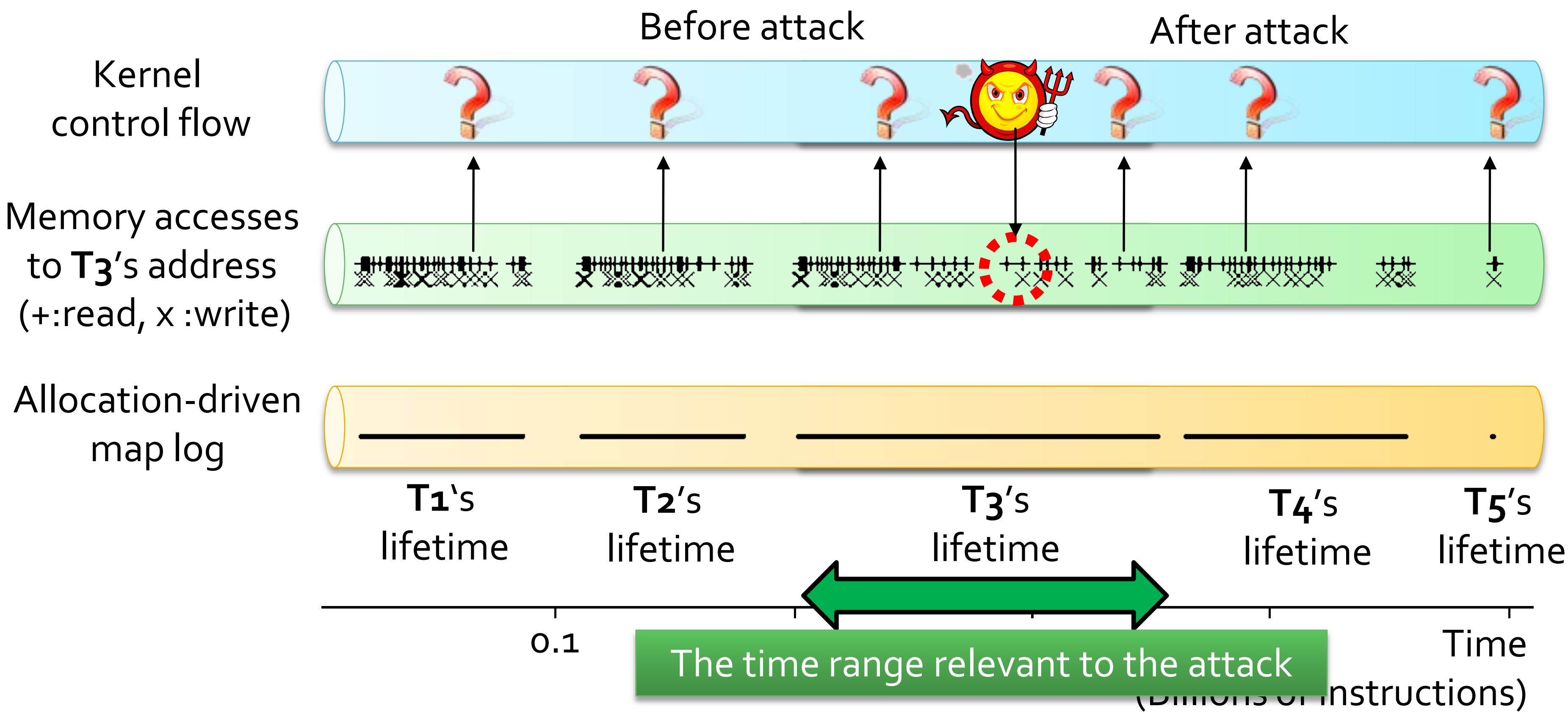
An application of the temporal view



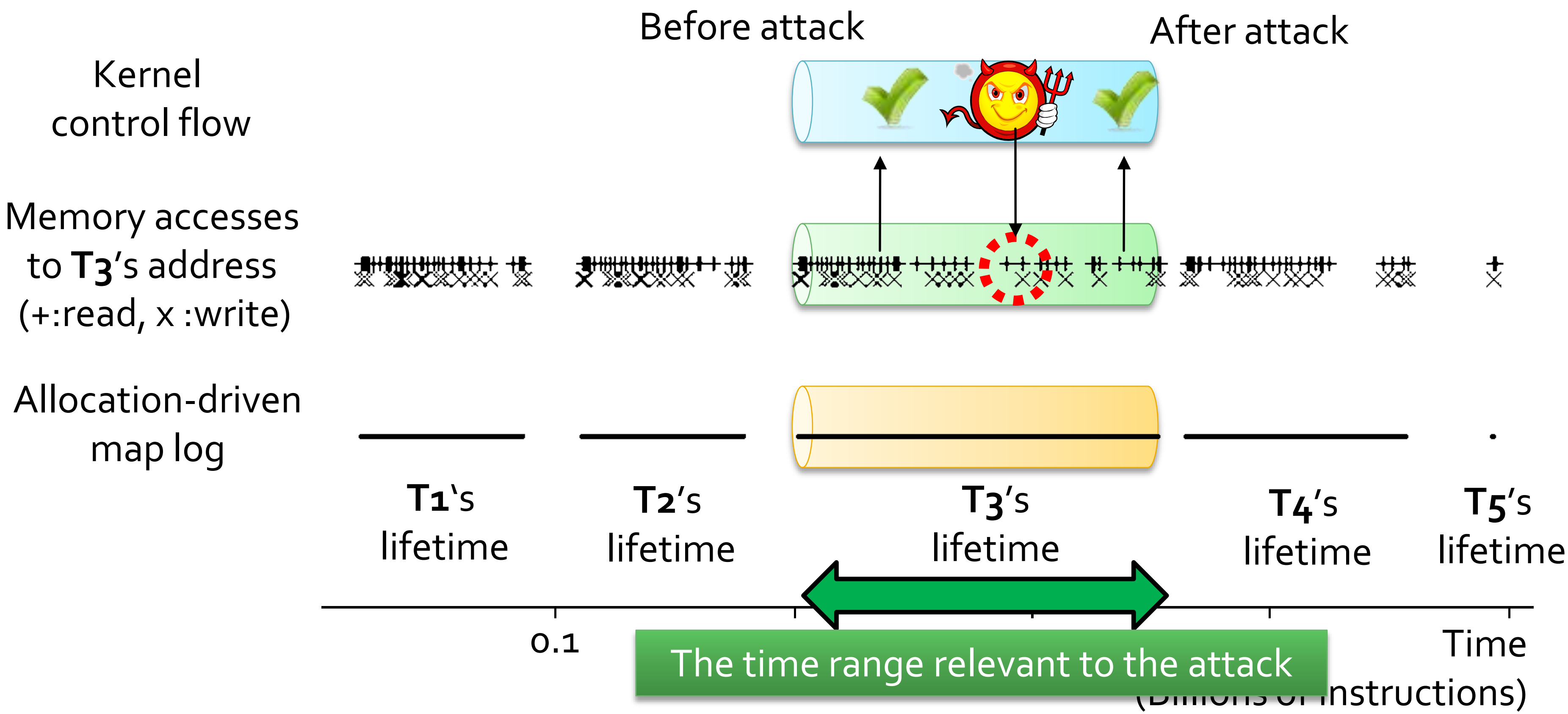
An application of the temporal view



An application of the temporal view

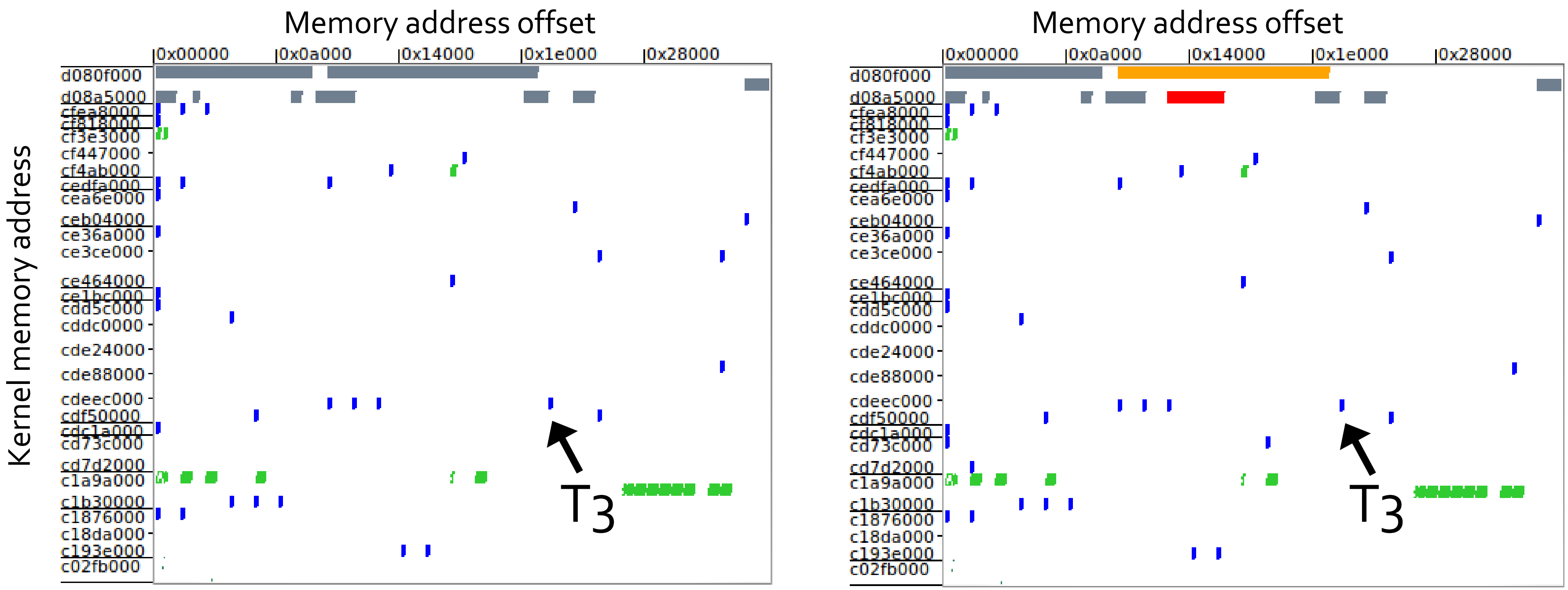


An application of the temporal view



- Malware analysis is guided to the attack victim objects (e.g., T3).

Malware analysis using a data view



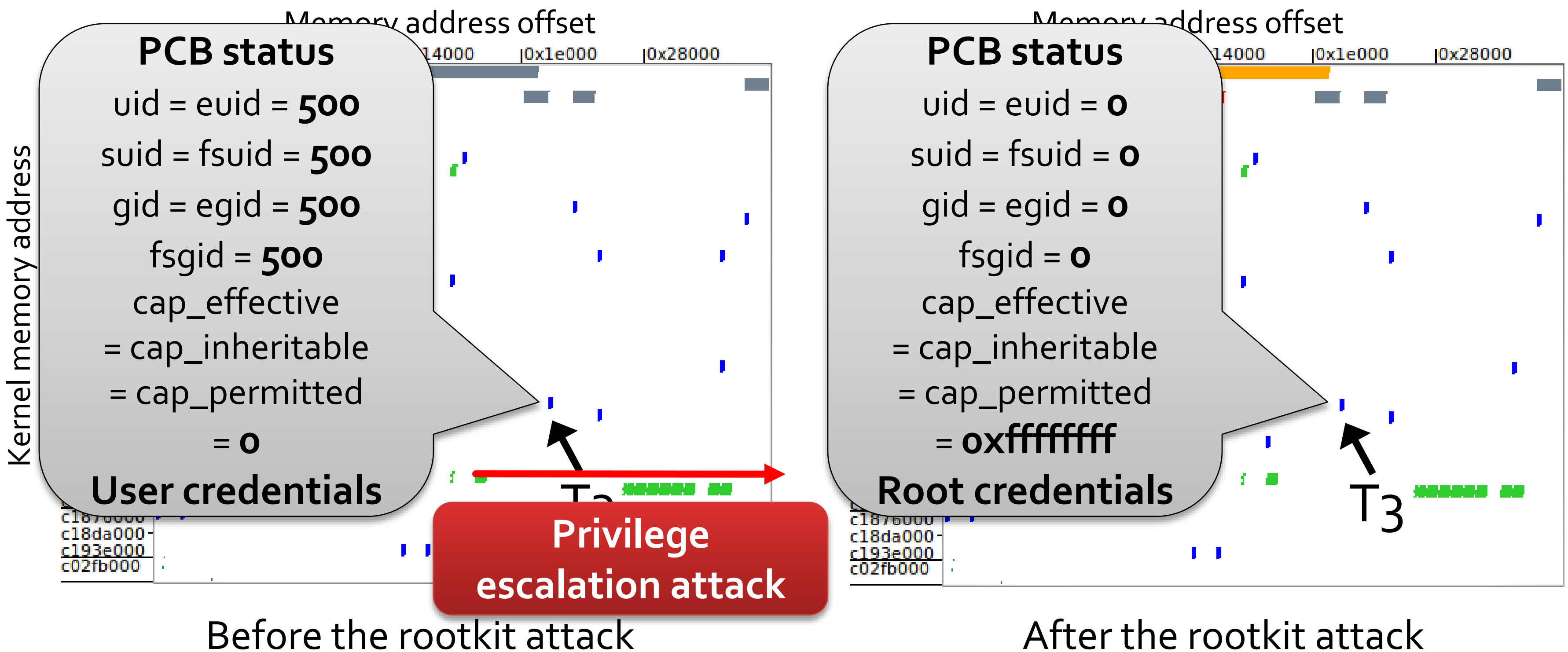
Before the rootkit attack

After the rootkit attack

Kernel object maps

task_struct (PCB) | proc_dir_entry | kernel modules | rootkit | ext3

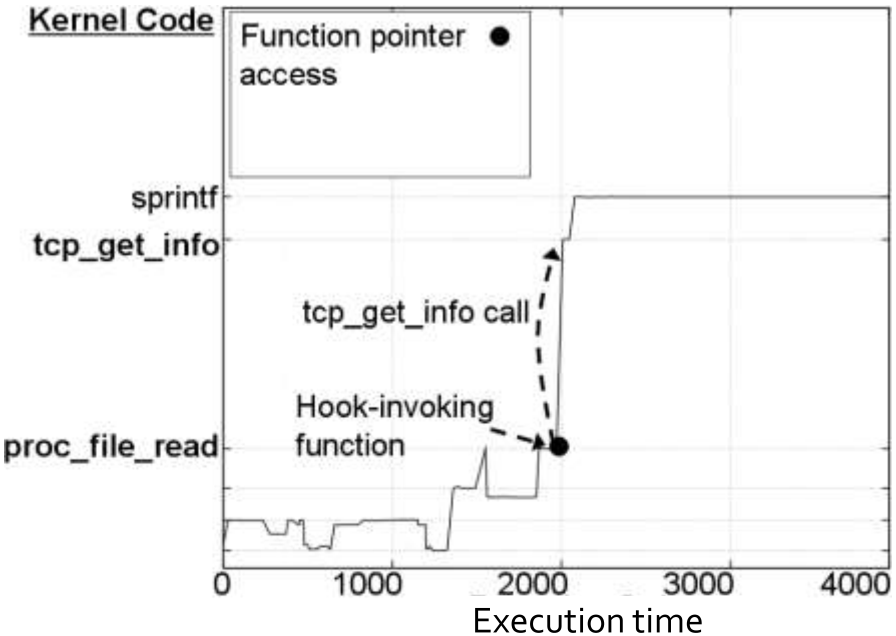
Malware analysis using a data view



Kernel object maps

task_struct (PCB) | proc_dir_entry | kernel modules | rootkit | ext3

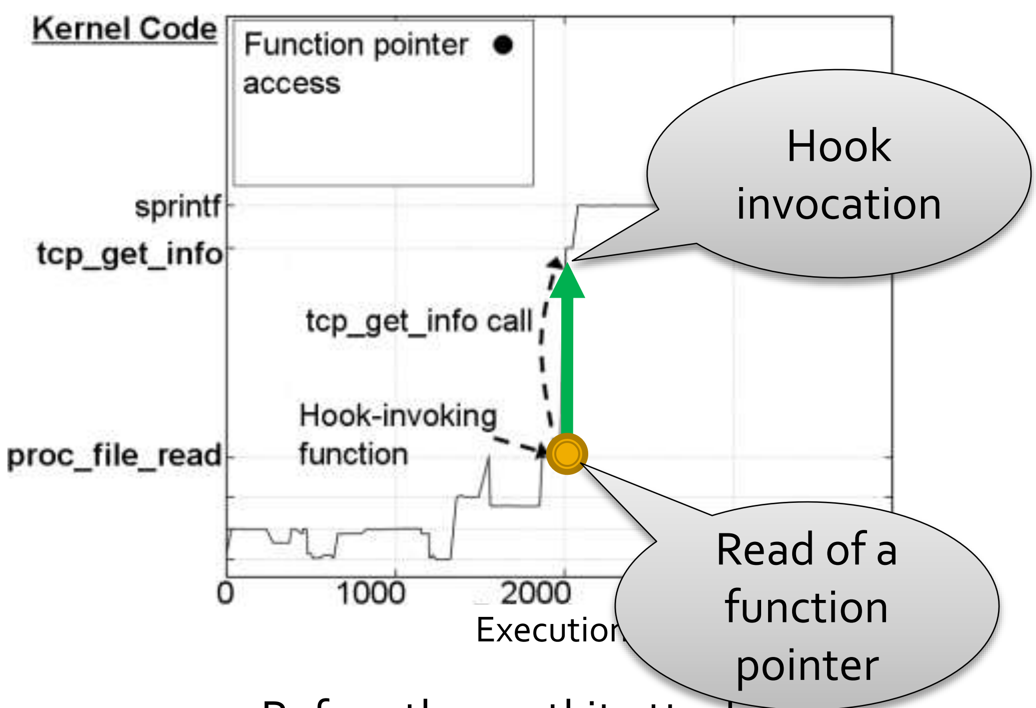
Malware analysis using a code view



Before the rootkit attack

Kernel control flow graphs

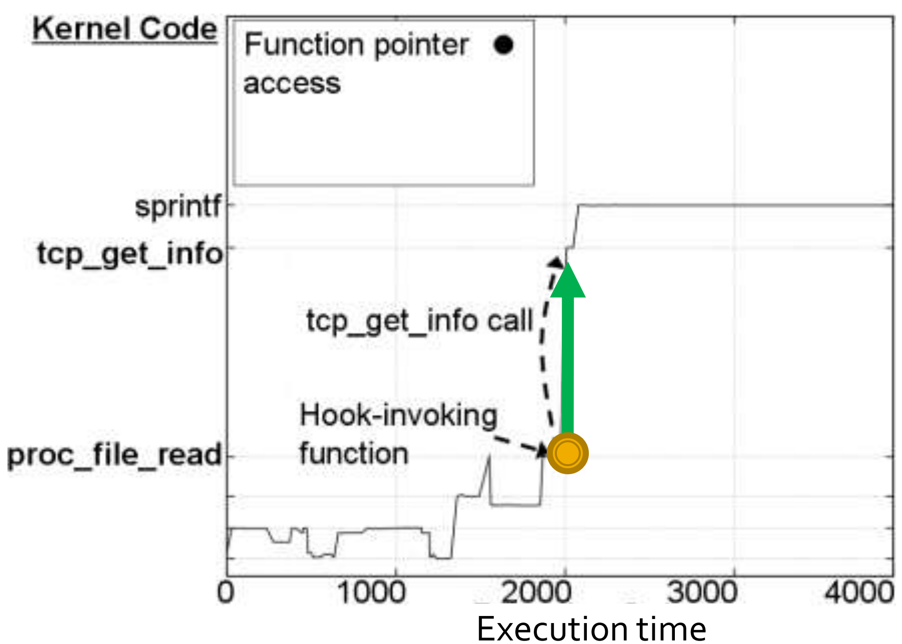
Malware analysis using a code view



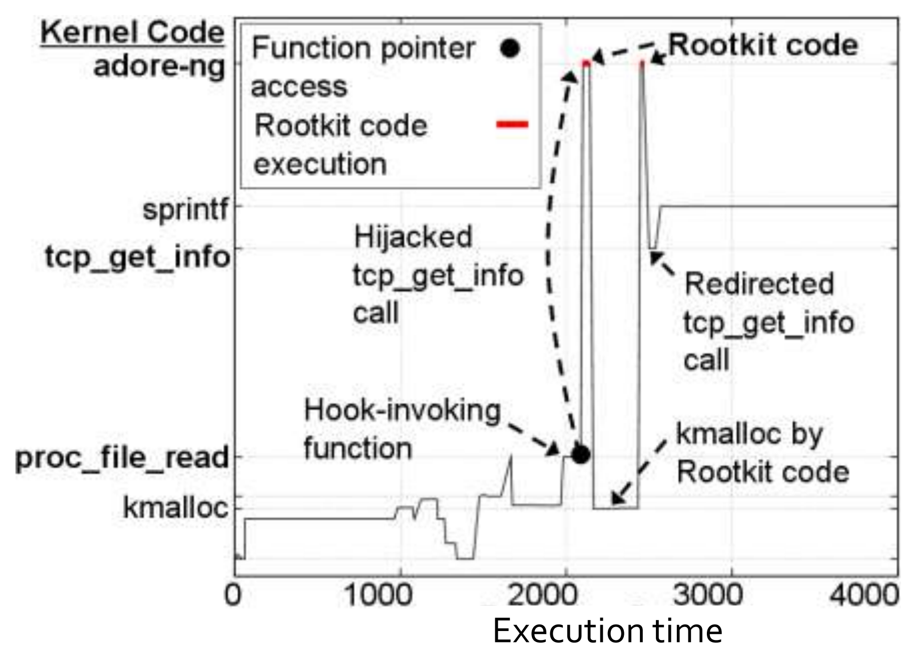
Before the rootkit attack

Kernel control flow graphs

Malware analysis using a code view



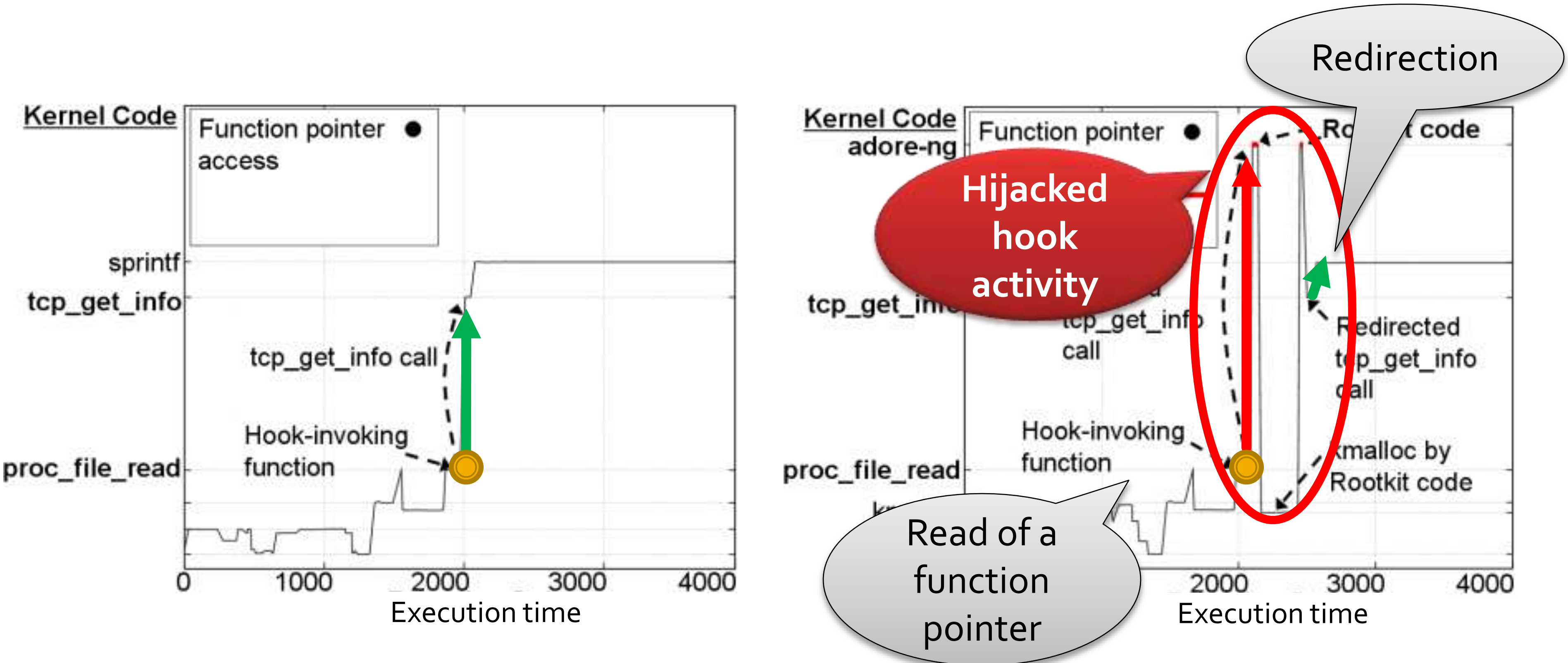
Before the rootkit attack



After the rootkit attack

Kernel control flow graphs

Malware analysis using a code view



Before the rootkit attack After the rootkit attack

Kernel control flow graphs

Discussions

- Memory objects of 3rd party drivers, malware
  - Source code is required to derive data types.
- Memory aliasing (type casting)
  - Allocation-driven map does not have aliasing problem by avoiding the evaluation of pointers.
  - Allocation using generic pointers : 0.1% of total objects
- Attack cases towards memory functions

Conclusion

- Un-tampered and temporal views of dynamic kernel objects can be enabled for malware analysis.
  - Kernel data hiding attacks can be detected by using an un-tampered view.
  - Temporal view can guide a malware analyzer to attack victim objects by tracking data lifetime.

Demo

- Main technique: Live kernel object map
  - Live status is dumped to a GUI every 5 seconds.
  - Dynamic changes of the map are illustrated.
- Applications: Hidden PCB and module detector
  - HP rootkit hides processes.
  - `modhide` rootkit hides kernel modules (drivers).
  - Data hiding attacks are checked every 5 seconds.
- URL: http://www.cs.purdue.edu/homes/rhee/pubs/raid2010_livedm.avi
- Note: some parts of a video clip are trimmed to reduce its play time.

Thank you,
Questions?