HookScout: Proactive Binary-Centric Hook Detection

Presenter: Lok Yan[1]
AFRL/RITA, Rome, NY 13441, USA
Heng Yin[1], Pongsin Poosankam[2,3], Steve Hanna[2],
and Dawn Song[2]

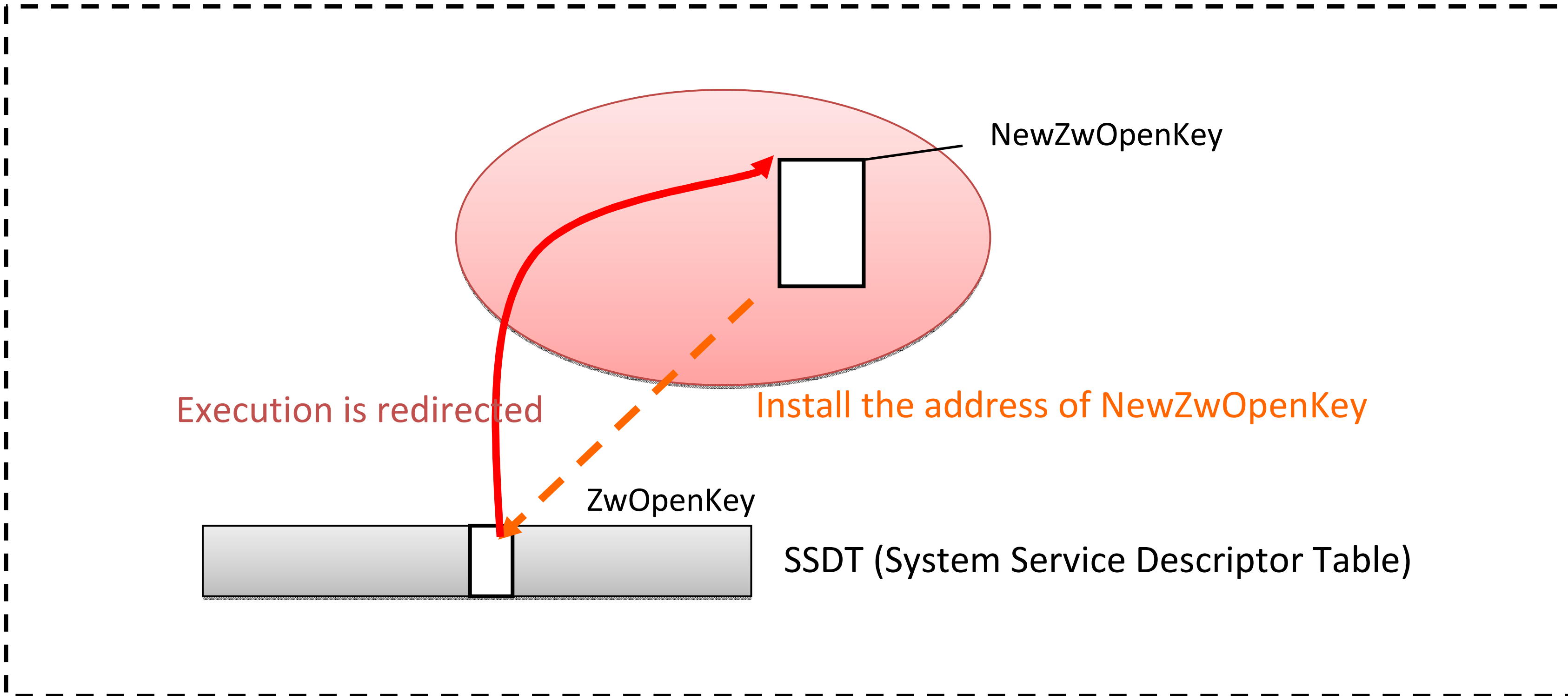
[1]Syracuse University
Syracuse, NY 13104
USA

[2]UC Berkeley
Berkeley, CA 94720
USA

[3]Carnegie Mellon University
Pittsburgh, PA 15213
USA

What is hook?

- Malware registers its own function (i.e. hook) into the target location
- Later, data in the hook site is loaded into EIP, and the execution is redirected into malware's own function.



an example of SSDT hooking

Hooking is an important attack vector

- malware often needs to install hooks to implement illicit functionalities
  - Rootkits want to intercept and tamper with critical system states
  - Network sniffers and stealth backdoors intercept network stack
  - Spyware, keyloggers and password thieves need to know when sensitive info arrives

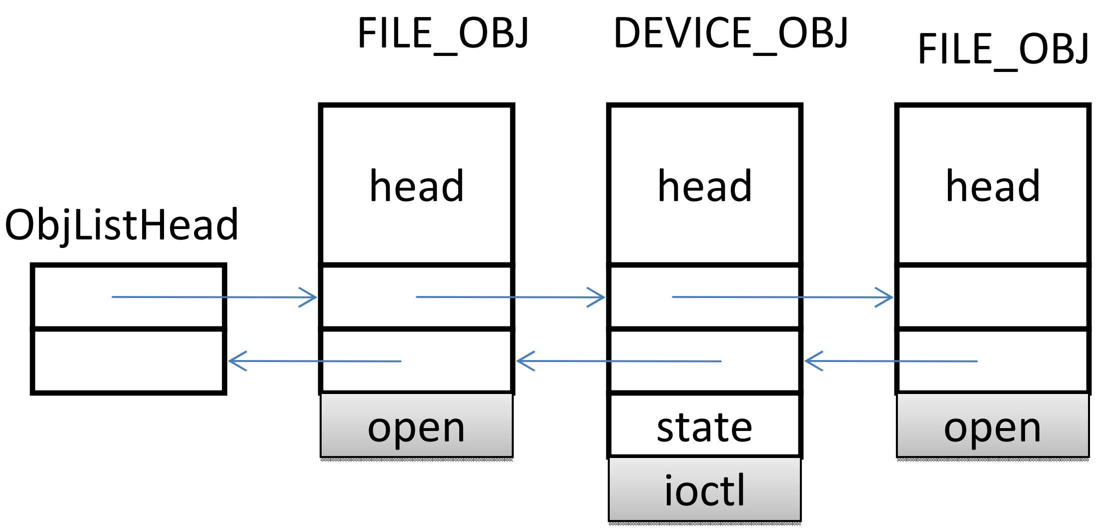
Hooking Techniques Are Evolving

- Old Technique: SSDT, IDT, IAT, EAT, etc.
  - Defeated by many existing hook detection tools
- New trend: function pointers in kernel data structures
  - IO completion routines
  - APC queues
  - Threads saved context
  - Protocol Characteristics Structures
  - Driver Object callback pointers
  - Timers
  - DPC kernel objects
  - DPC scheduled from ISR
  - IP Filter driver hook
  - Exception handlers
  - Data buffer callback routines
  - TLS callback routines
  - Plug and play notifications
  - All kinds of WDM driver stuff
  - **Many more, …**

Advantages of Function Pointer Hooking

- Attack space is vast
  - ~20,000 function pointers in Windows kernel
- Hard to locate and validate
  - ~7,000 in dynamically allocated memory regions
  - Many of them in polymorphic data structures
  - A polymorphic hash table in Windows kernel

Example: A polymorphic linked list

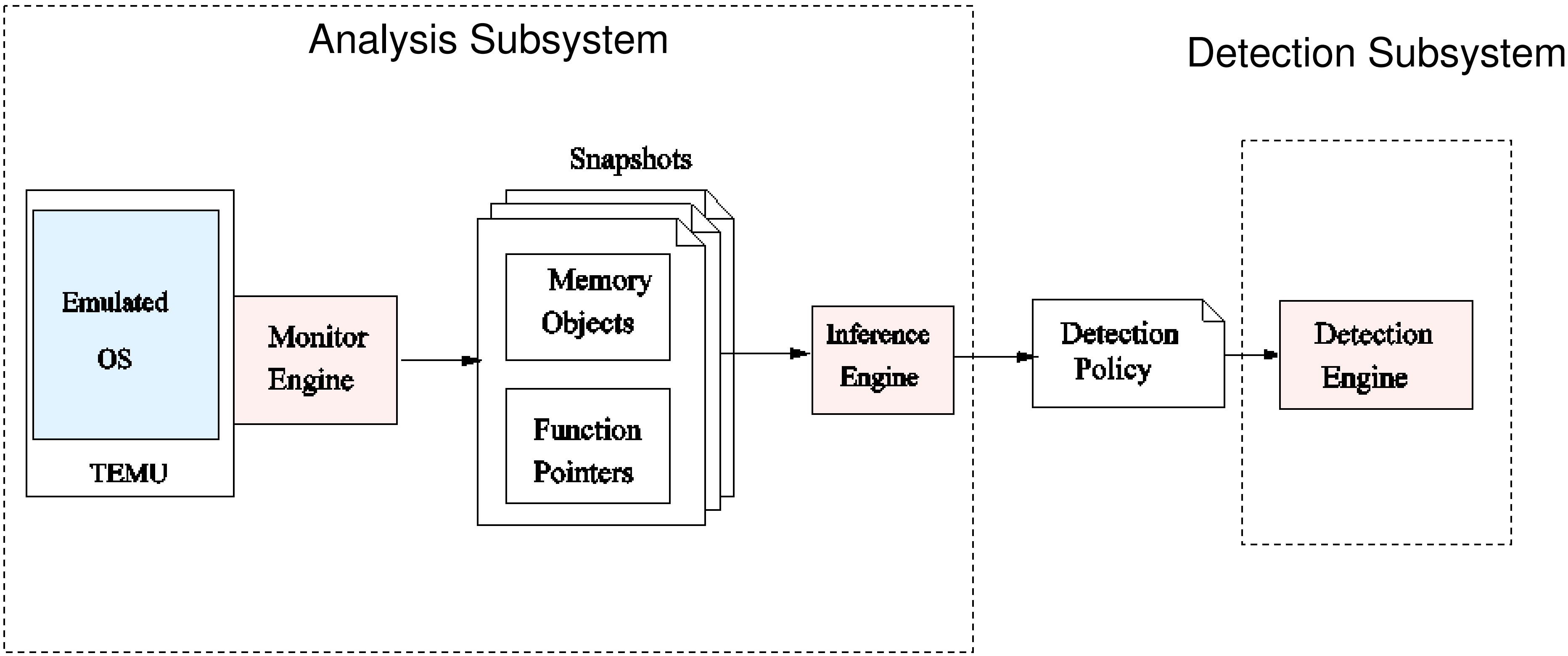



```
typedef struct {
  OBJ_HEAD head;
  LIST_ENTRY link;
  int (*open)(char *n, char *m);
  ...
} FILE_OBJ;

typedef struct {
  OBJ_HEAD head;
  LIST_ENTRY link;
  int state;
  int (*ioctl)(char *buf, int size);
  ...
} DEVICE_OBJ;

LIST_ENTRY ObjListHead;
```

Our Goal

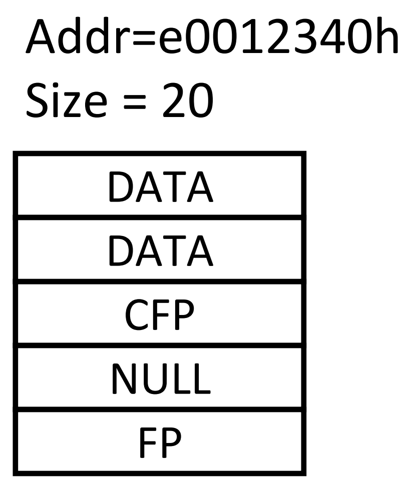
- Given the binary distribution of an OS kernel, automatically generate a hook detection policy
  - Locate function pointers
    - Deal with polymorphic data structures
  - Validate function pointers
    - only 3% ever change in their lifetime (from our analysis)
    - Simple policy: check if constant function pointers ever change

System Overview



Monitor Engine

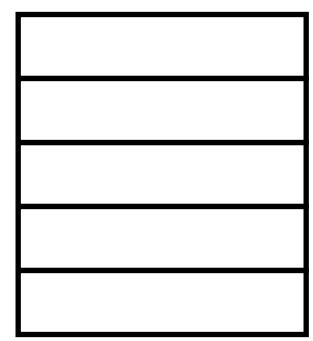
- Goal: determine concrete memory layout
  - For each static/dynamic memory object, determine primitive types for each memory word
  - Primitive types: NULL, FP, CFP, DATA
- Solution:
  - Monitor memory objects
  - Track function pointers



Monitor Engine: Monitor Memory Objects

- Run the guest OS within TEMU
  - TEMU: a whole-system binary analysis platform, based on QEMU
- For dynamic objects: Hook memory allocation/deallocation routines
  - ExAllocatePoolWithTag, ExFreePool
  - RtlAllocateHeap, RtlFreeHeap
- For static objects: Hook module loading routine
  - MmLoadSystemImage

Addr=e0012340h
Size = 20



Monitor Engine: Track Function Pointers

```
CreateFile()
{
  FILE_OBJ *f = malloc(sizeof(FILE_OBJ));
  …
  f->open = MyFileCreate;
  InsertListTail(&f->link, &ObjListHead);
  …
}
```

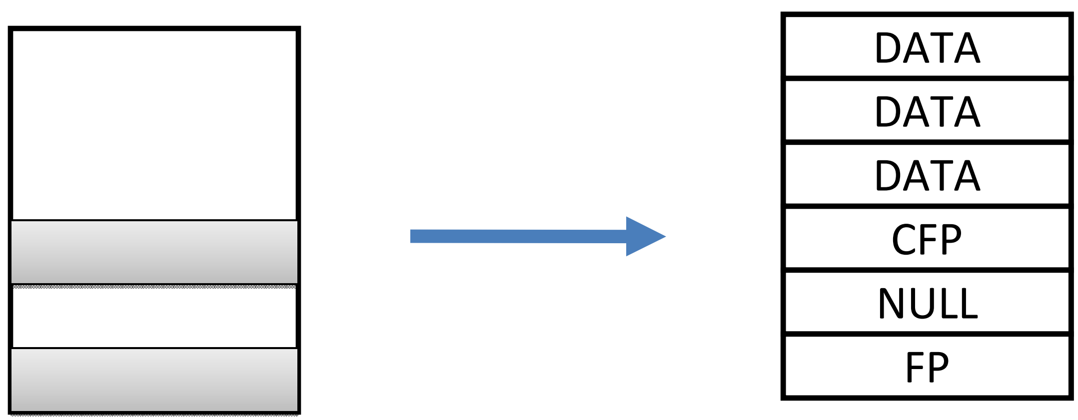
1. Hooked RtlAllocateHeap

```
804d7200: call malloc
…
804d7230: mov [ebp-50h], 805d5141h
```

2. IDA Pro plugin processes Relocation and Import Address Tables
3. Identifies and taints initial function pointers

Addr=e0012340
Size = 40
Caller=804d7200

Addr=e0012340
Size = 40
Caller=804d7200

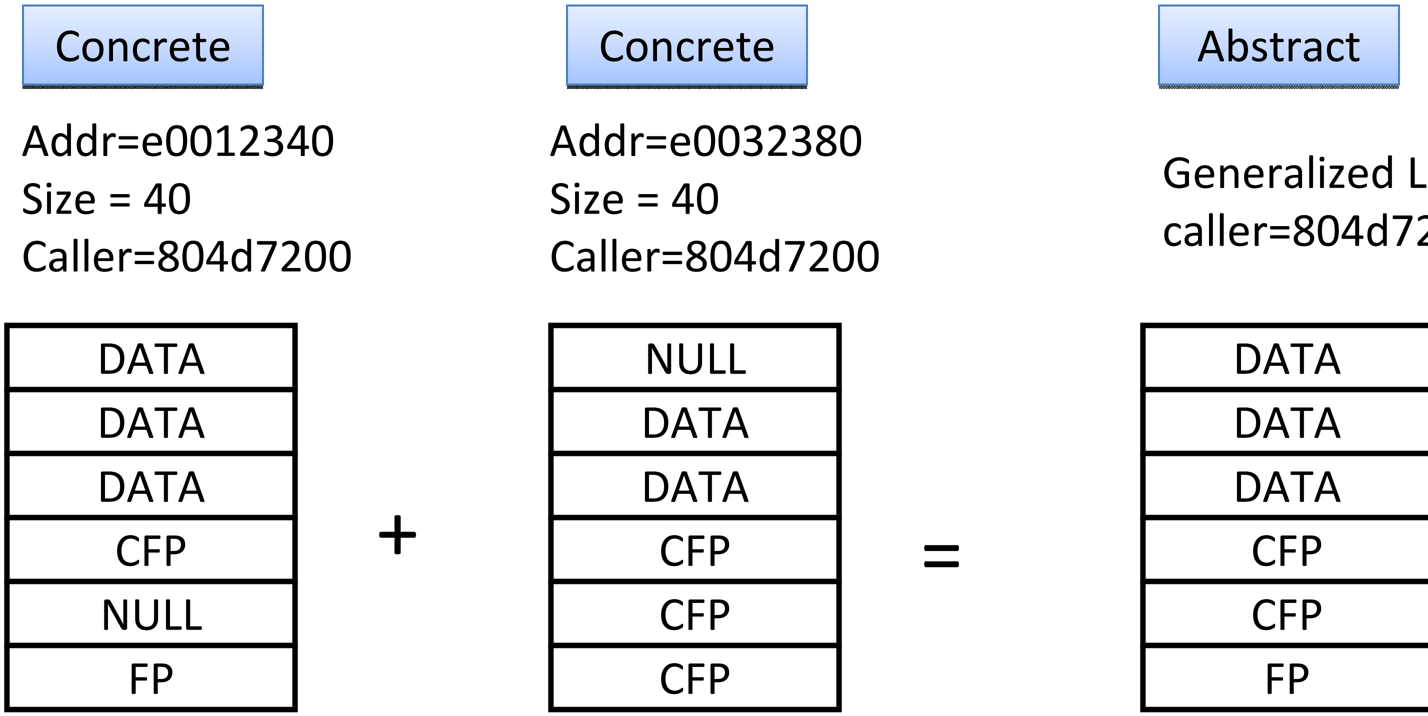


"Concrete Layout"

Inference Engine

- **Goal**: Infer abstract memory layout
- **Approach**: context-sensitive abstraction
  - **Notion**: Object creation context is the execution context where an object is created (e.g., caller of malloc)
    - Binary point of view: return addresses on the call stack
  - **Rationale**: Objects created under the same context have the same type
  - **Solution**: Merge concrete layouts with the same context into an abstract layout

Inference Engine: Context-Sensitive Type Inference



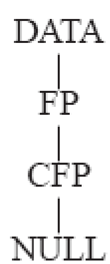


Figure 3: Lattice for join operation ⊔

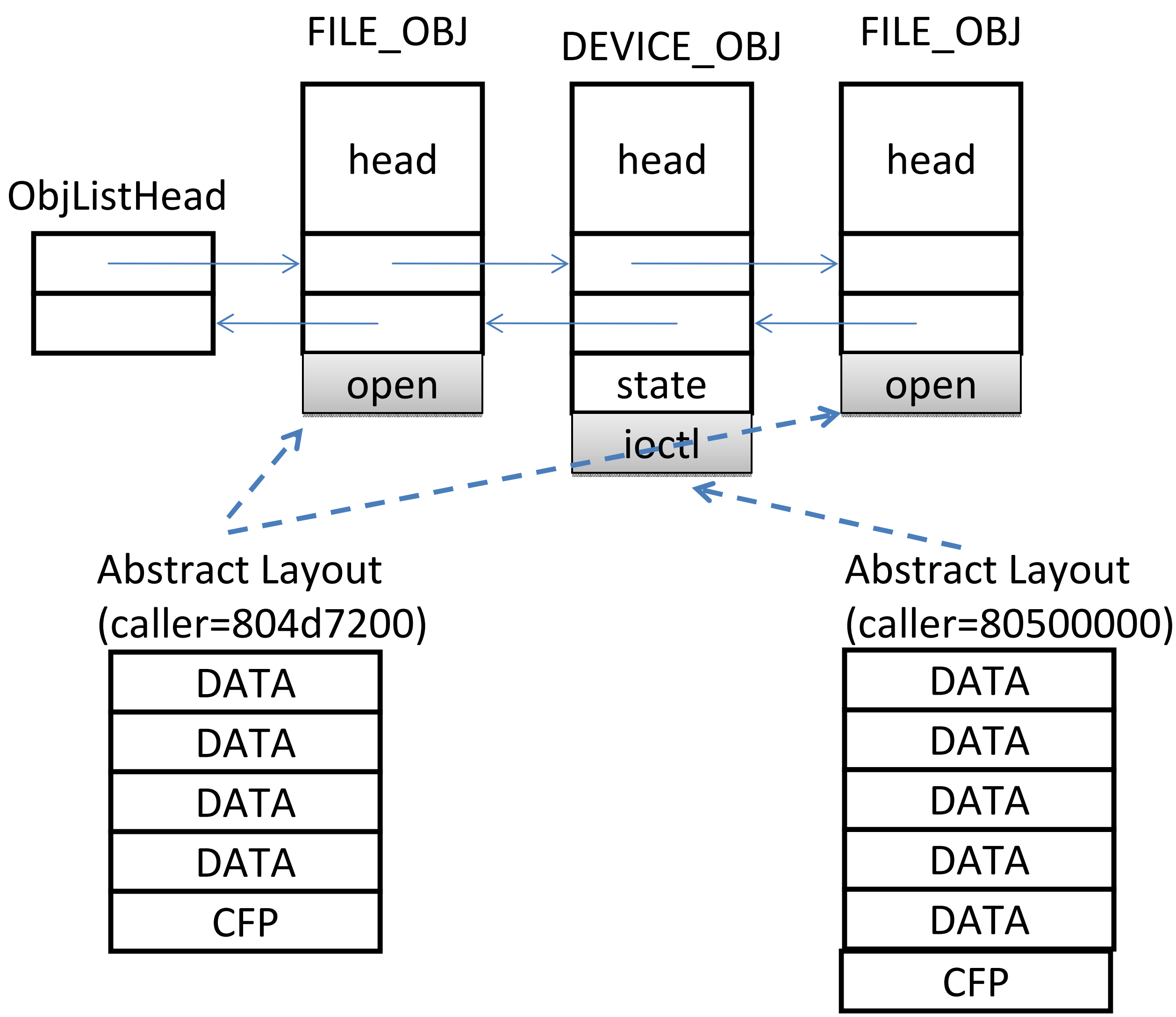
⊔	NULL	CFP	FP	DATA
NULL	NULL	CFP	FP	DATA
CFP	CFP	CFP	FP	DATA
FP	FP	FP	FP	DATA
DATA	DATA	DATA	DATA	DATA

Table 1: Matrix for join operation ⊔

Detection Engine

- Goal:
  - Enforce the hook detection policy on user's machine
- Solution:
  - Monitor memory objects
    - Hook the same set of functions
  - Apply the abstract layout
    - Use the return addresses as the key to the abstract layout
- Implementation:
  - Kernel module vs. Hypervisor

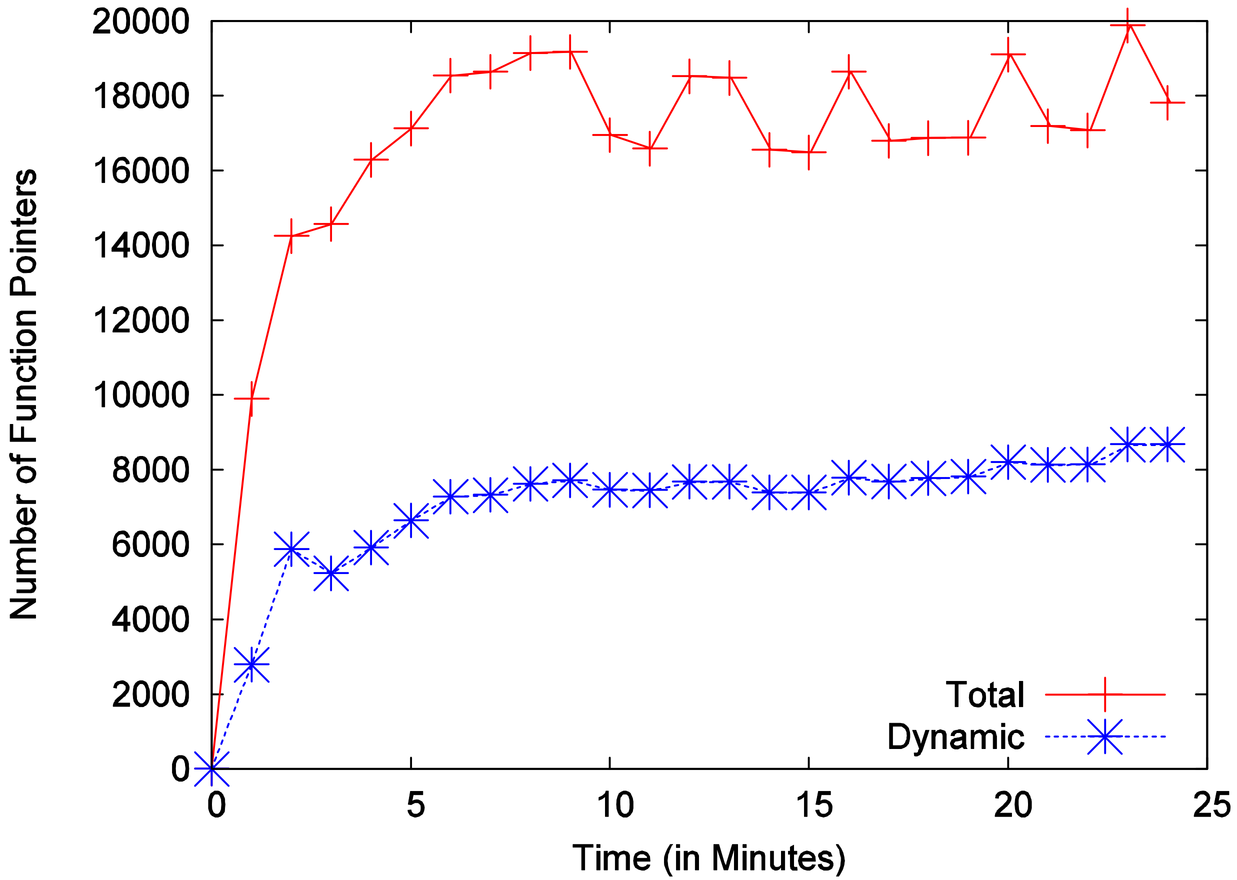
Detection Engine: go back to the example



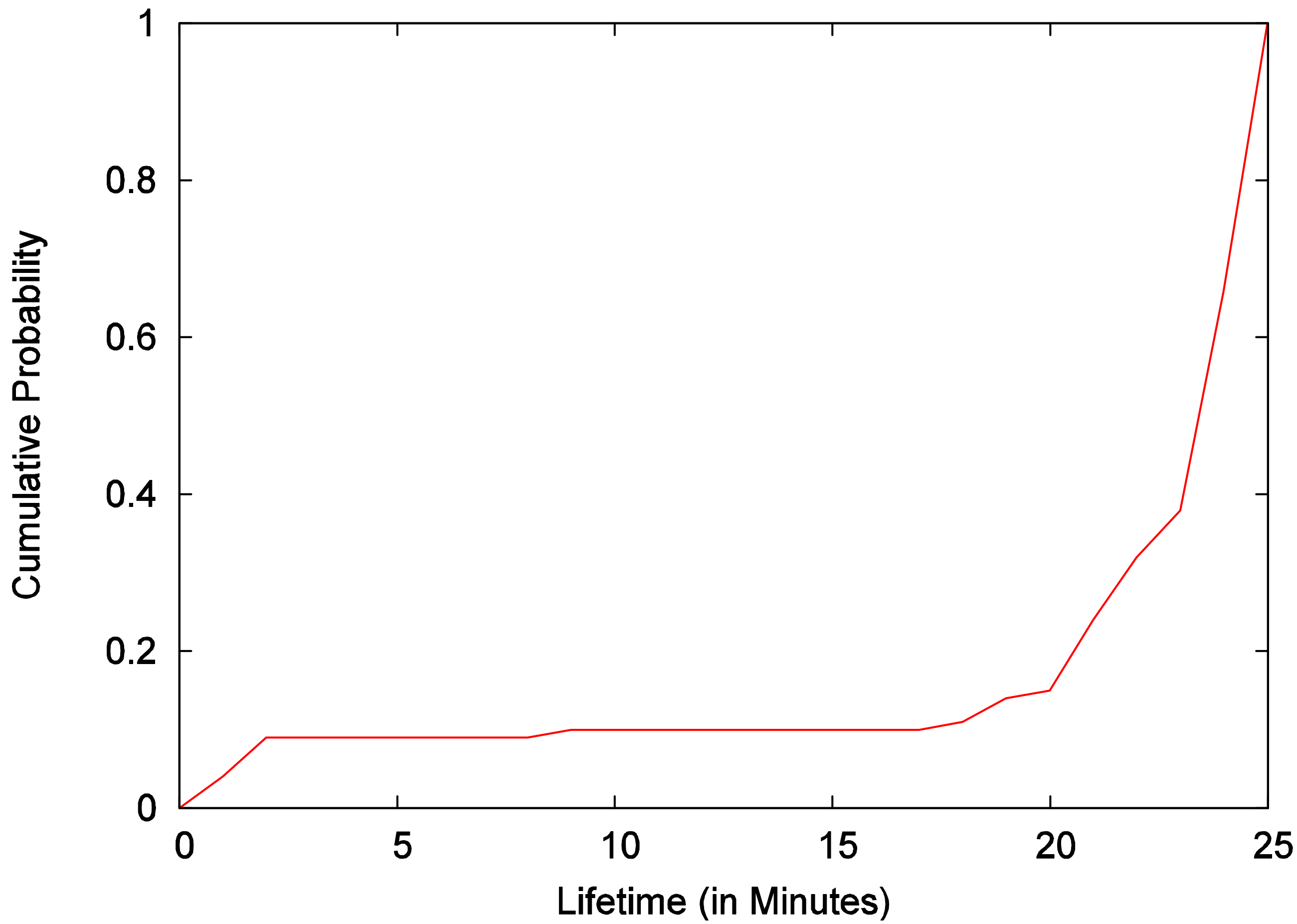
Experimental Evaluation

- Aspects to Evaluate
  - Attack Space
  - Analysis subsystem: policy coverage
  - Detection subsystem:
    - realworld rootkits/performance/false alarms
- Experimental Setup
  - Host machine: 3.0GHz CPU 4 GB RAM Ubuntu
  - Guest machine: 512MB RAM Windows XP SP2

Evaluation: Attack Space



Evaluation: Function Pointer Lifetime Distribution



Evaluation: Policy Generation

Level	Coverage		Templates	
	AVG	STDEV	Raw	Final
1	94.67%	2.97%	3518	308
2	96.10%	1.92%	4285	405
3	96.74%	1.64%	5270	511

Experimental Setup:

•Total of three 25 minute runs, a snapshot every 15 seconds

•Runs 1 and 2 used to generate abstract templates policy

•For each snapshot in Run 3

Coverage = Number of Function Pointers identified by Policy / Total number of Function Pointers

•Level indicates context sensitivity, i.e. # of return addresses

Policy Generation Performance: 70 seconds / snapshot, ~4hours for 200 snapshots

Evaluation: Realworld Rootkit Detection

Sample Name	Hooking Region	IceSword [12]	VICE [3]	RAIDE [19]	HookScout
HideProcessHookMDL [21]	SSDT	✓	✓	✓	✓
Sony Rootkit [27]	SSDT	✓	✓	✓	✓
Storm Worm [28]	SSDT	✓	✓	✓	✓
Shadow Walker [21]	IDT	?	✓	✓	✓
basic_interrupt_3 [21]	IDT	?	✓	✓	✓
TCPIRPHOOK [21]	Tcp driver object	×	✓	✓	✓
Rustock.C [22]	Fastfat driver object	×	×	✓	✓
Uay Backdoor [29]	NDIS data block	×	×	✓	✓
Keylogger-1	Static data region for keyboard driver	×	×	×	✓
Keylogger-2	Dynamic data region for keyboard driver	×	×	×	✓

Table 5: Detection Results

Evaluation: Performance of Detection Subsystem

Workload	w/o HookScout	w/ HookScout		Slowdown	
		1s	5s	1s	5s
Boot OS	19.43 s	20.70s	20.43 s	6.5%	5.1%
Copy directories	7.57 s	8.09s	7.68 s	6.9%	1.5%
(De)compress files	23.84 s	24.44s	23.51 s	2.5%	-1.4%
Download a file	23.59 s	24.49s	24.42 s	3.8%	3.5%

* No false alarms were raised during the testing period

Limitations

- Coverage – what if people exploit the 5% that is not covered?
- Detection Interval – is 5s or even 1s frequent enough?
- Uncommon Proprietary Device Drivers – HookScout utilizes QEMU and since other proprietary drivers are never installed, they are not analyzed.
- Limited test cases for the dynamic analysis
- Kernel module can be subverted or mislead – A hypervisor is preferable

Related Work

- **Post-mortem Analysis**
  - K-Tracer
  - PoKeR
- **Proactive Defense – Prevent Untrusted Code Execution**
  - Livewire
  - SecVisor
  - Patagonix
- **Proactive Defense - Control Flow Integrity**
  - SBCFI
  - Gibraltar
  - SFPD
  - HookMap
  - HookSafe

Conclusion

- Function pointer hooking is a new trend
  - Large attack space
  - Hard to detect
  - Without OS source code, even harder
- We developed HookScout
  - Binary-centric: deal with OS binary code
  - Context-sensitive: deal with type polymorphsim
  - Proactive: detect attacks in advance