Ensuring Operating System Kernel Integrity with OSck

Owen Hofmann, Alan Dunn, Sangman Kim, Indrajit Roy*, Emmett Witchel

UT Austin
*HP Labs

Rootkits are dangerous

- Adversary exploits insecure system
  - Leave backdoor to facilitate long-term access
- A real world problem
  - Malware involved in breach of 95% of data records [Verizon Data Breach Report 2010]
  - 85% installed backdoors
- Why are rootkits such a pain?

Rootkits are difficult to detect

- Key behavior: hide system state to conceal presence

- Files
  - Conceal suspicious control / configuration files

- Processes
  - Conceal backdoor login process
  - In Unix, a special case of file hiding in /proc

- Other system state
  - Open network ports
  - Loaded kernel modules

Kernel rootkits even more so

- User-level vectors detectable
  - Kernel will still report correct state
  - Hash system binaries

- Kernel rootkits can be undetectable by users
  - Attacker has access to kernel memory
  - Modify kernel state to hide resources
  - Kernel reports incorrect state to *all* user programs

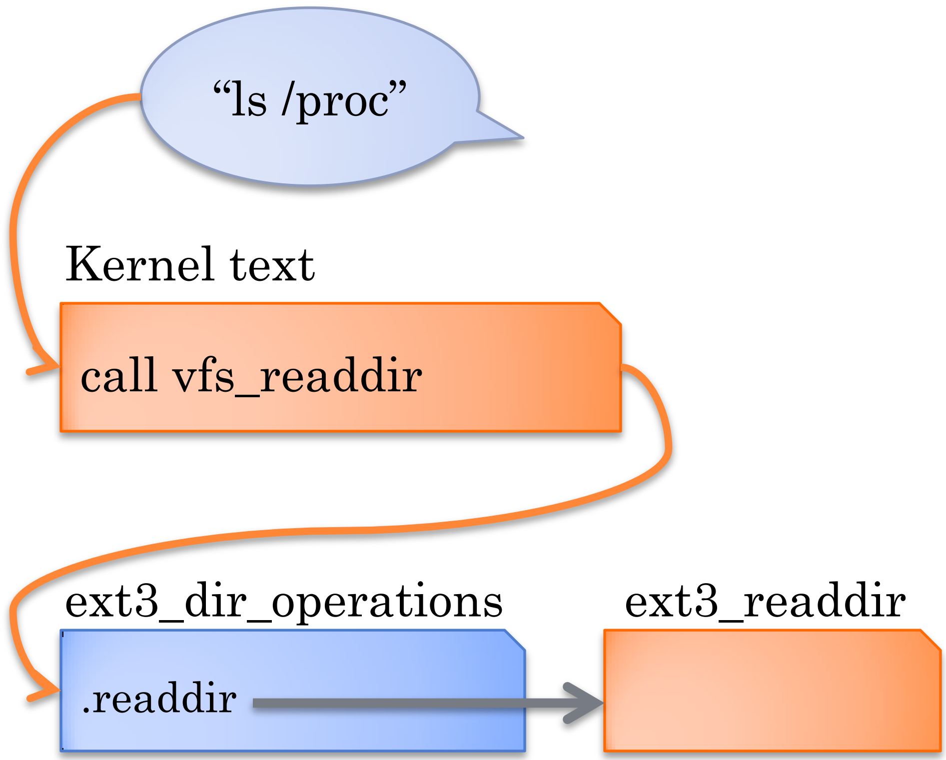
- Modify kernel control flow or data
  - Violate some kernel *invariant*

Rootkits change control flow

- Modify functions for examining system state

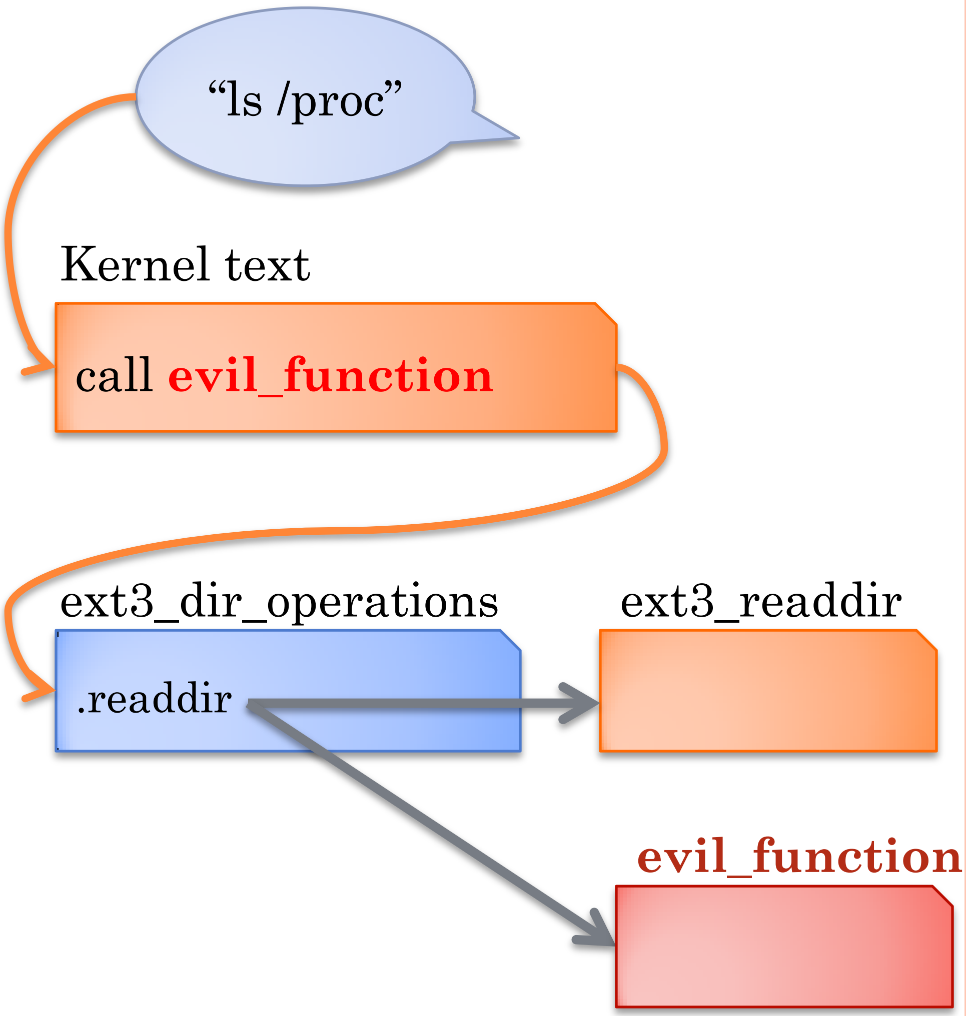
- Kernel text
  - Change instructions
  - Invariant: text is immutable

- Function pointers
  - In mutable data memory
  - Invariant: pointers point to one of a few valid entry points



ROOTKITS CHANGE CONTROL FLOW

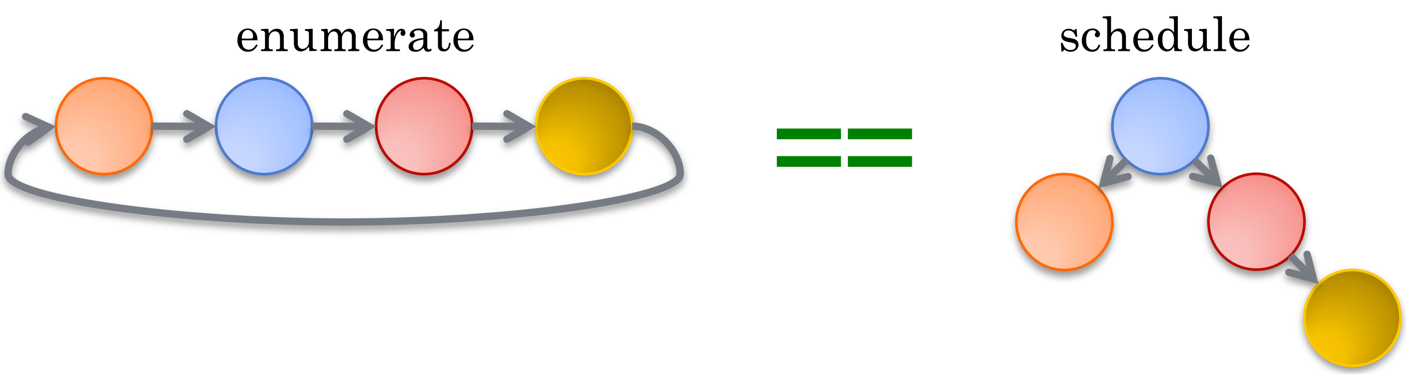
- Modify functions for examining system state
- Kernel text
  - Change instructions
  - Invariant: text is immutable
- Function pointers
  - In mutable data memory
  - Invariant: pointers point to one of a few valid entry points



Rootkits change data structures

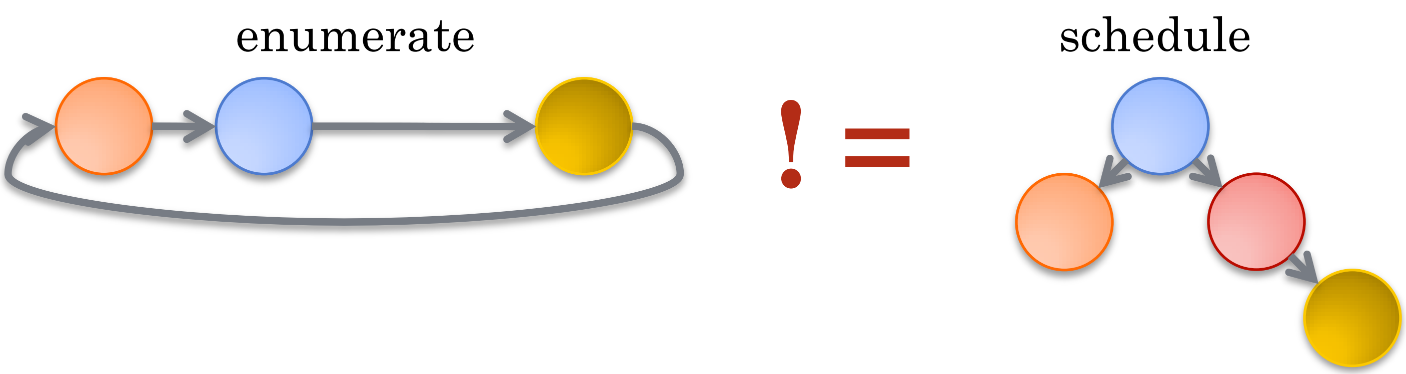
- Kernel assumes invariants hold between data structures
  - Linux: tree for scheduling, list for enumerating processes
  - Invariant: structures represent same set

- Rootkit can modify heap to hide state



Rootkits change data structures

- Kernel assumes invariants hold between data structures
  - Linux: tree for scheduling, list for enumerating processes
  - Invariant: structures represent same set
- Rootkit can modify heap to hide state

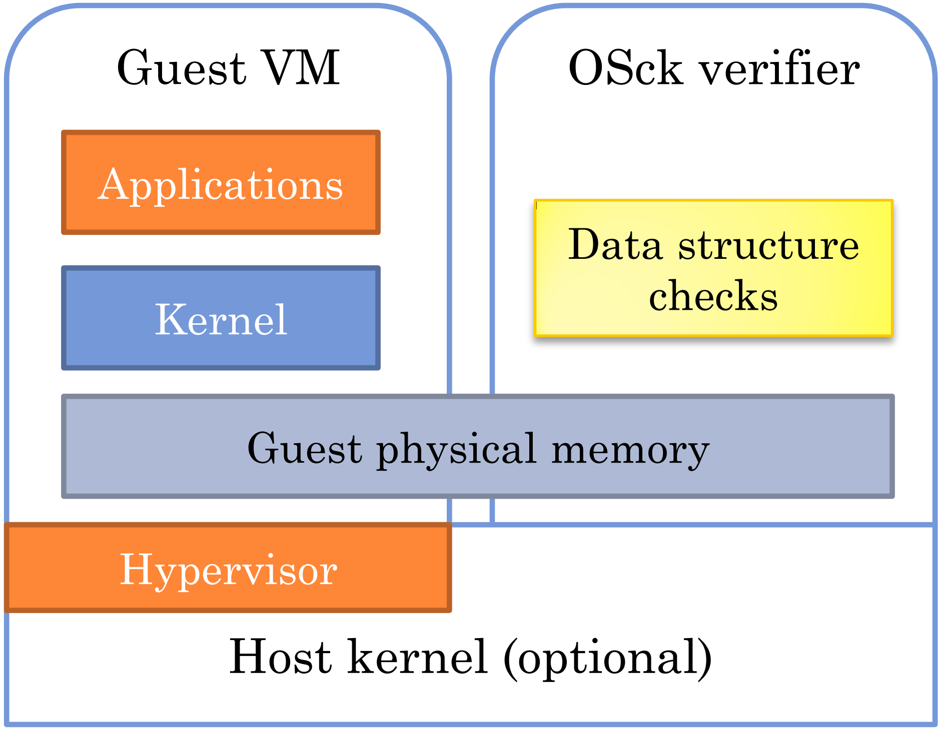


Protecting the kernel

- OSck: ensure kernel integrity by checking invariants
  - (It's like fsck)
- Identify key invariants subverted by rootkits
  - Control-flow
  - Important heap structures (e.g. process list)
- Generate code to check invariants
  - Automatic: analyze source code
  - Manual: write ad-hoc integrity checks
- Isolate checking code from operating system

OSck architecture

- Virtualize kernel
- Run verifier process alongside kernel
  - Has access to kernel compile-time information
- Hypervisor provides verifier access to kernel memory
- Periodically scan memory for violations
  - Configurable performance overhead



OSck design goals

- Efficiency and safety
  - Verifier must inspect all kernel memory
  - Use hints from untrusted kernel to speed checks
- Programmability
  - Not all checks are automatic
  - Make it easy to write ad-hoc checks
  - Source-to-source translation of kernel data structures
- Concurrency
  - Checking code runs concurrently with kernel
  - Safely handle concurrency-related errors

OSck design goals

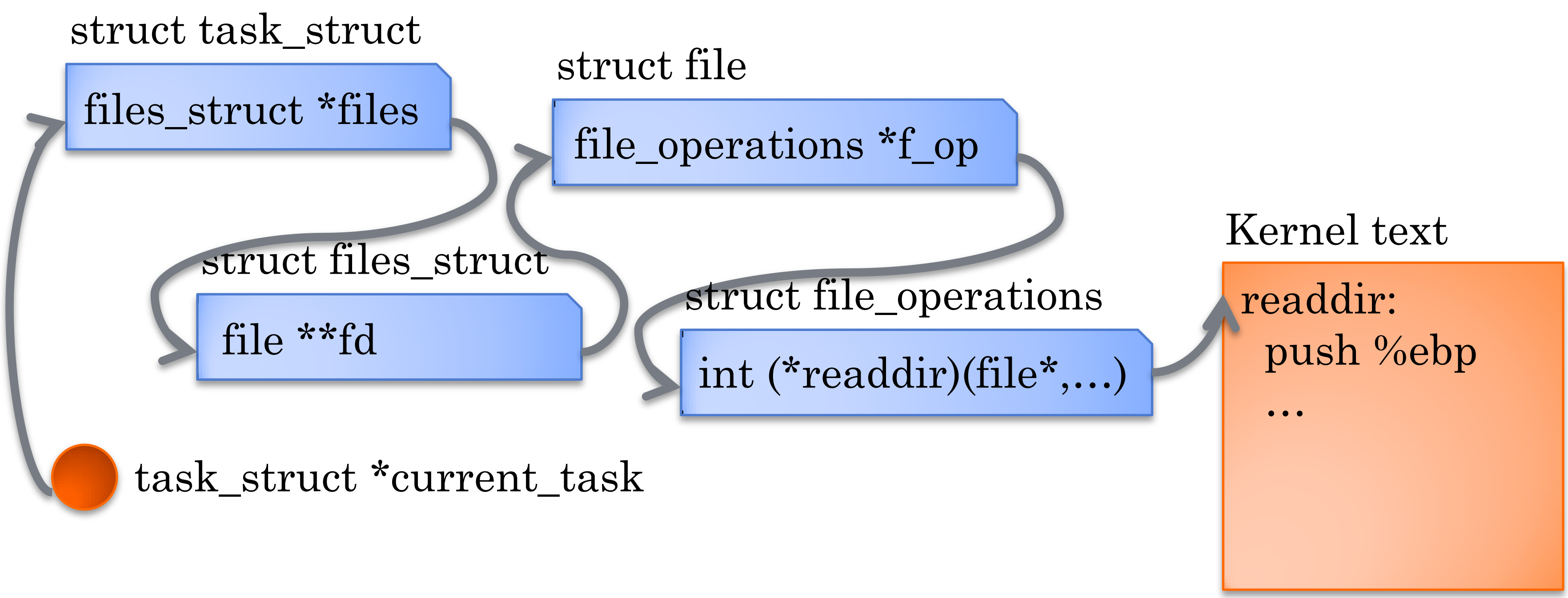
- Efficiency and safety
  - Verifier must inspect all kernel memory
  - Use hints from untrusted kernel to speed checks
- Programmability
  - Not all checks are automatic
  - Make it easy to write ad-hoc checks
  - Source-to-source translation of kernel data structures
- Concurrency
  - Checking code runs concurrently with kernel
  - Safely handle concurrency-related errors

PROTECTING CONTROL FLOW

- *Static* and *persistent*
  - Kernel text and processor state (e.g. IA32_LSTAR)
  - Protect text with hardware page protection
  - Disallow updates to special registers

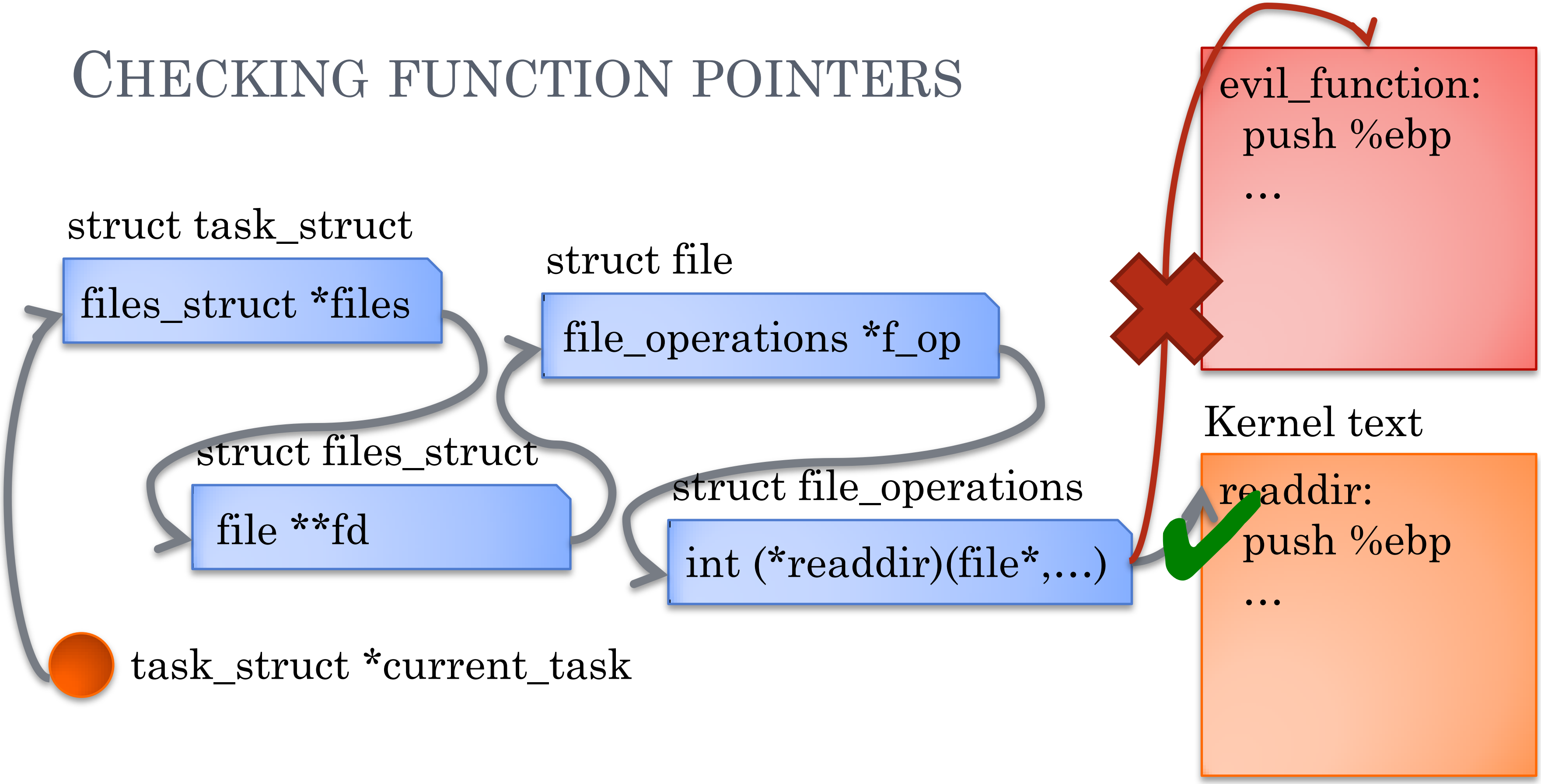
- *Dynamic*
  - Function pointers in data memory
  - Invariant: point to one of a few valid entry points
  - Can be at any memory address
  - Can be a variety of types

Checking function pointers



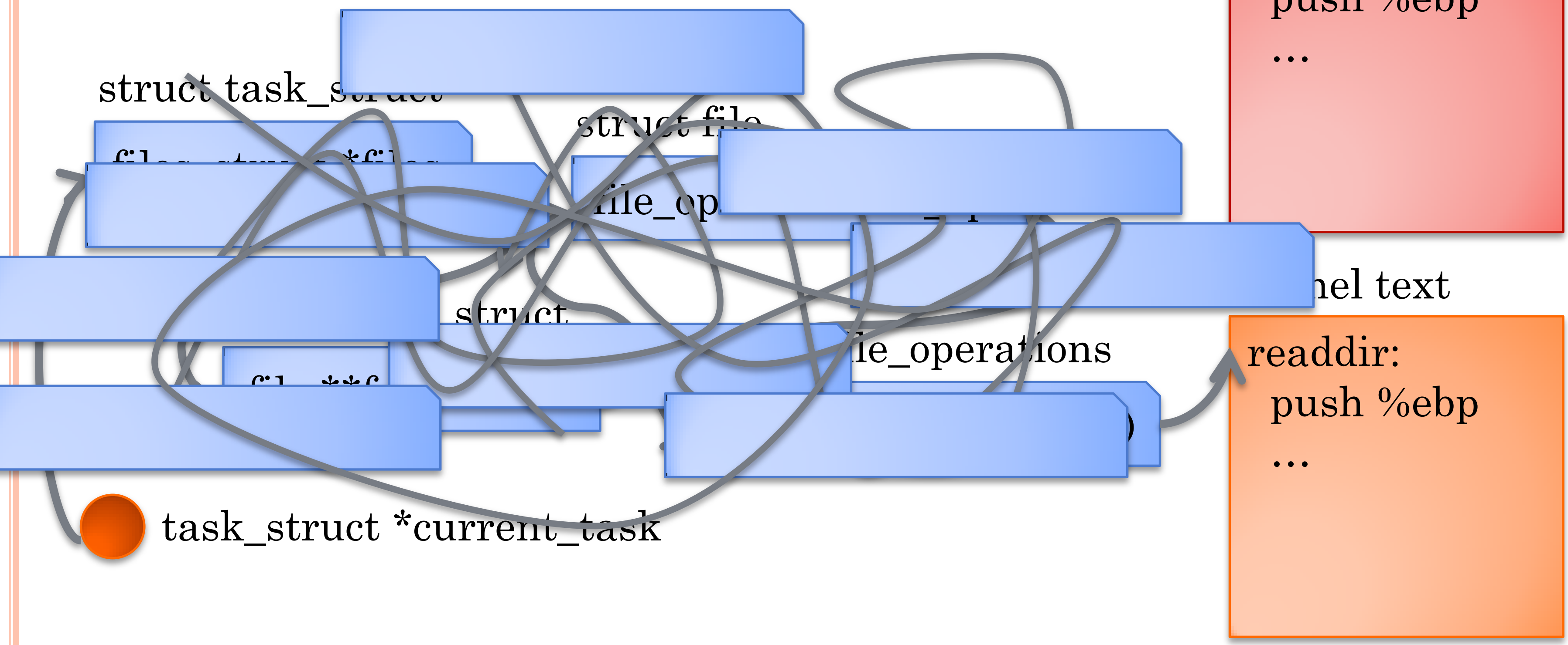
- How does kernel get to function pointer?
  - Start at global root (symbol)
  - Traverse graph of data structures

Checking function pointers



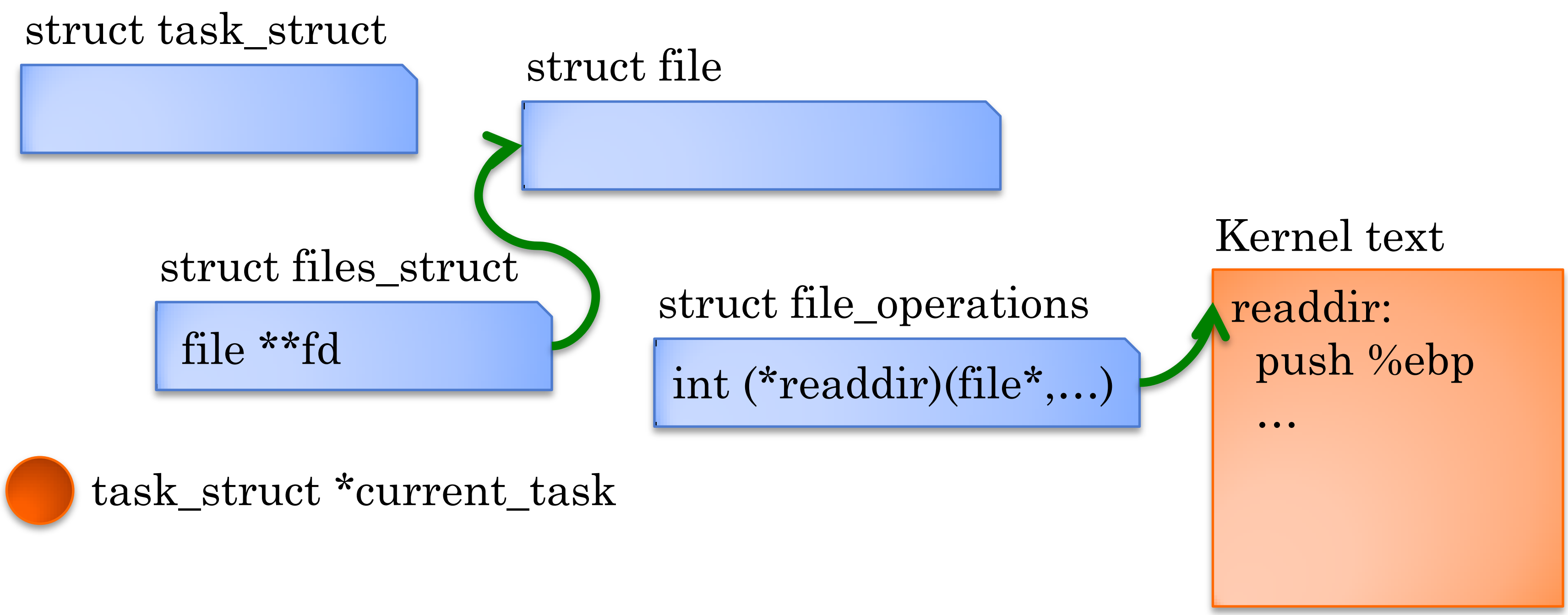
- *State-based control flow integrity* [Petroni & Hicks]
  - Start at global root (symbol)
  - Traverse graph of data structures
  - Ensure function pointers point to valid entry points

Checking function pointers



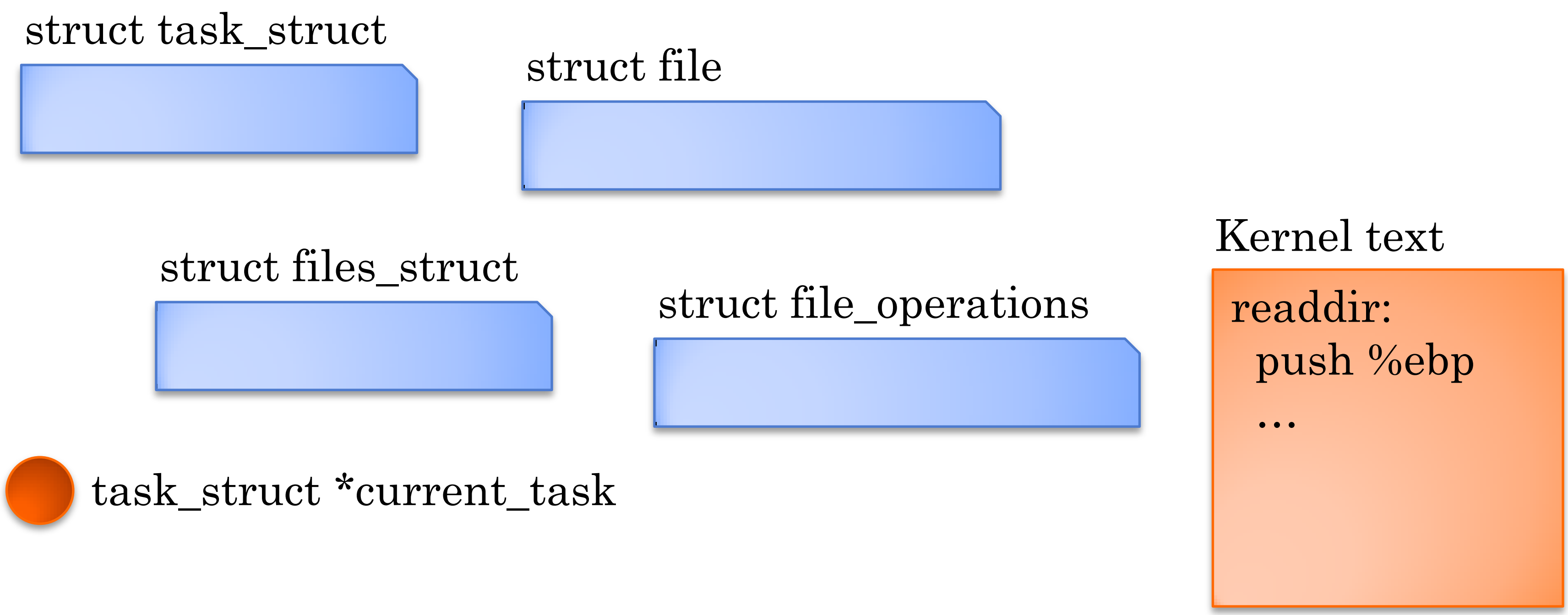
- Traversing large graphs is not great
  - Significant amount of dynamic state
  - Must avoid runaway pointers, etc.
  - We can do better

Checking with type information



- Map kernel memory to type
  - Pick an object (any object)
  - Verify its pointers
- Verify all kernel memory in single pass

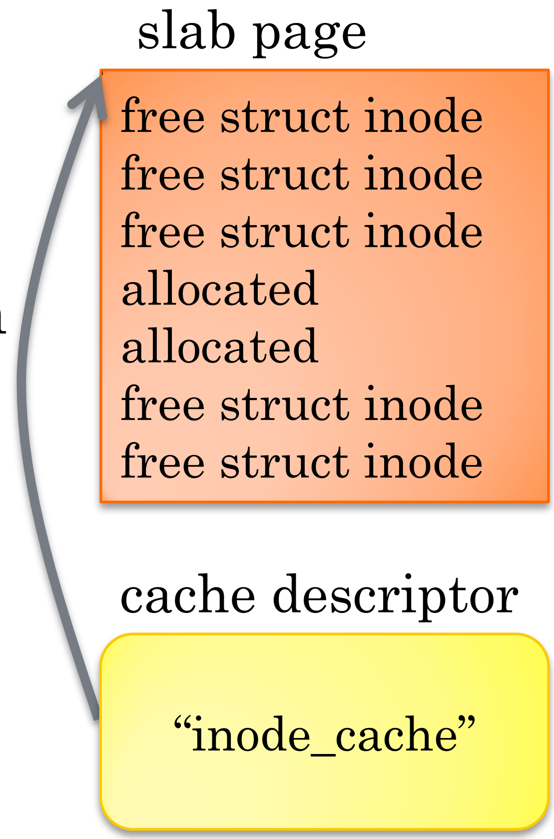
Checking with type information



- Where does type information come from?
  - Kernel: allocates memory

LINUX SLAB ALLOCATION

- Kernel allocates memory with *caches*
  - Per-type allocators
  - Objects of same type on same page
- Source analysis associates cache with type
  - Identify allocation sites, allocated types
- OSck reads kernel page metadata
  - Determine cache for each page
  - Objects on page have cache's type

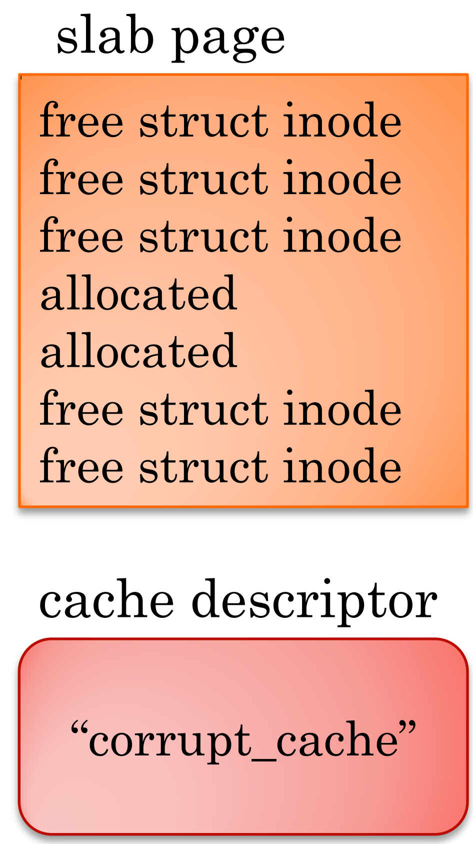


Linux slab allocation

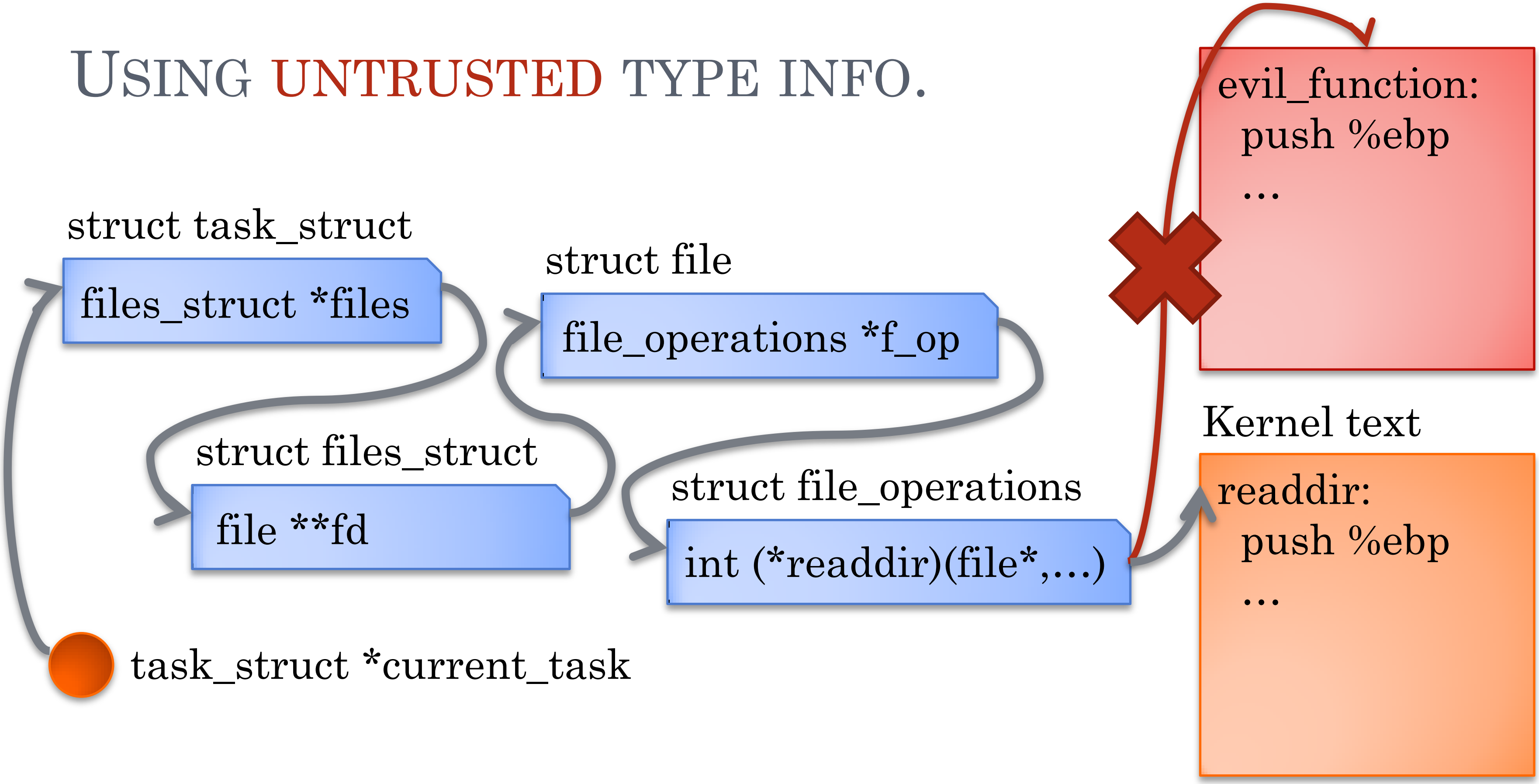
- Kernel allocates memory with *caches*
  - Per-type allocators
  - Objects of same type on same page

- Source analysis associates cache with type
  - Identify allocation sites, allocated types

- OSck reads kernel page metadata
  - Determine cache for each page
  - Objects on page have cache's type

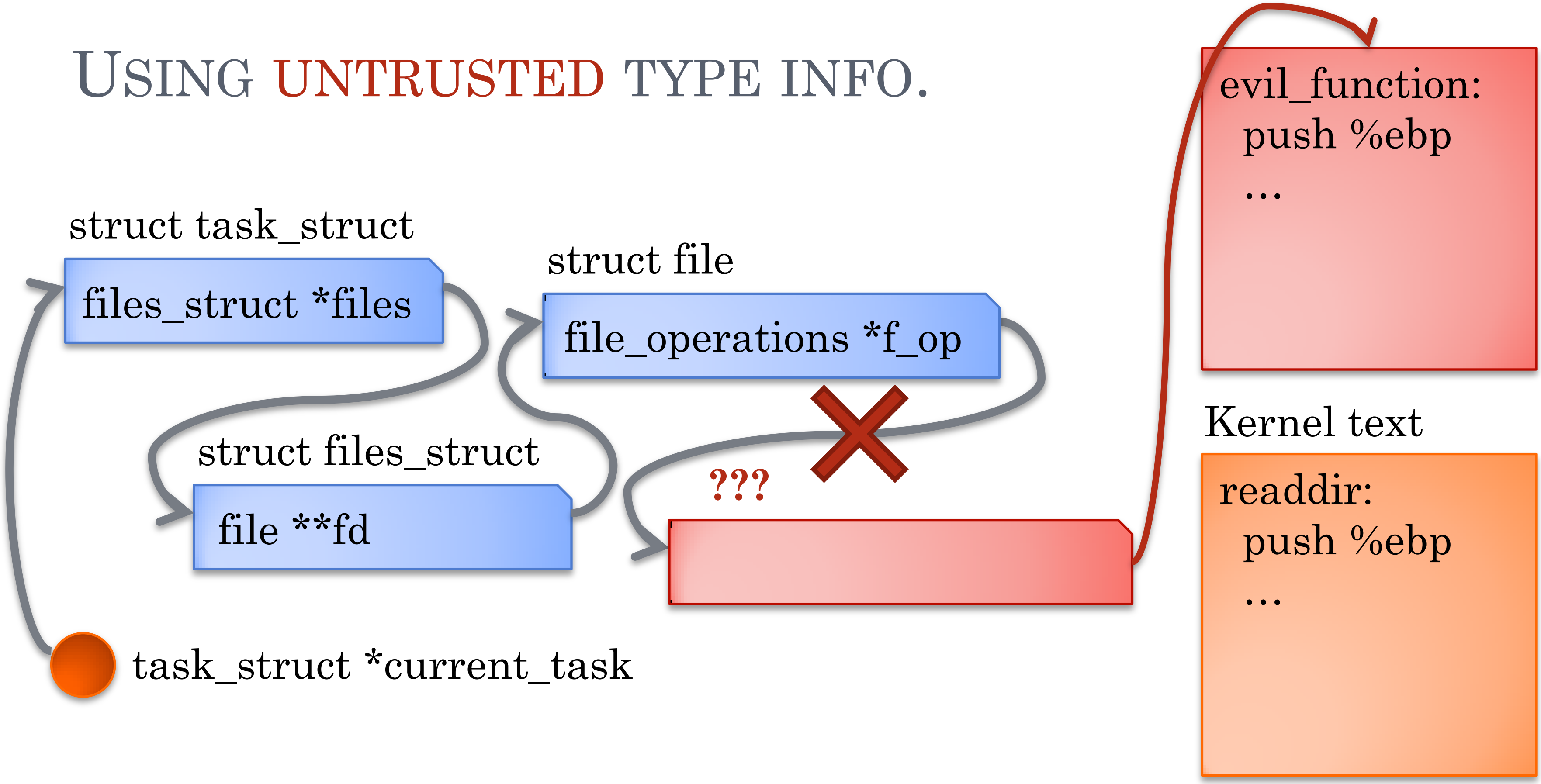


USING UNTRUSTED TYPE INFO.



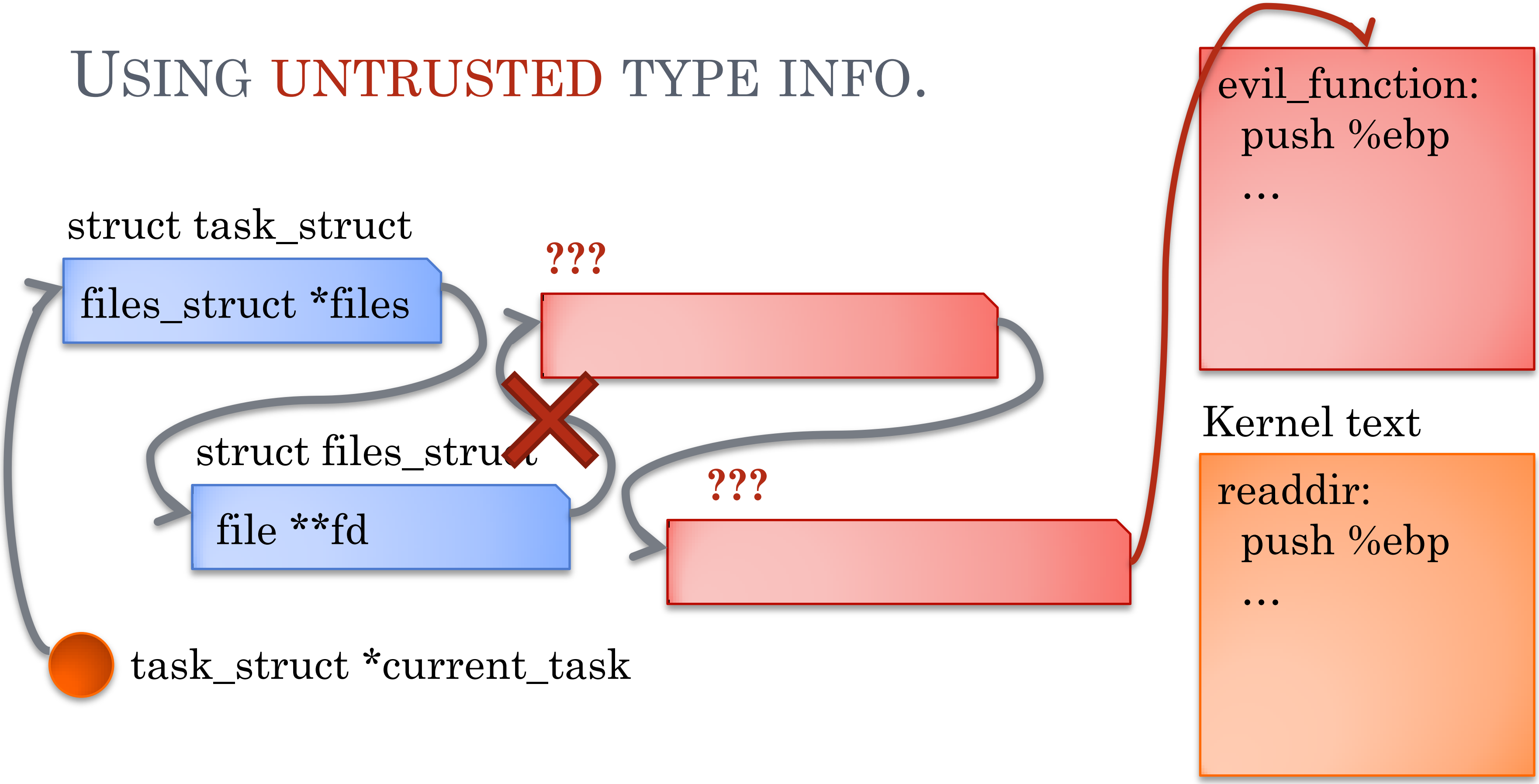
- Cannot change type assigned to function
  - Valid entry points determined at compile time

Using untrusted type info.



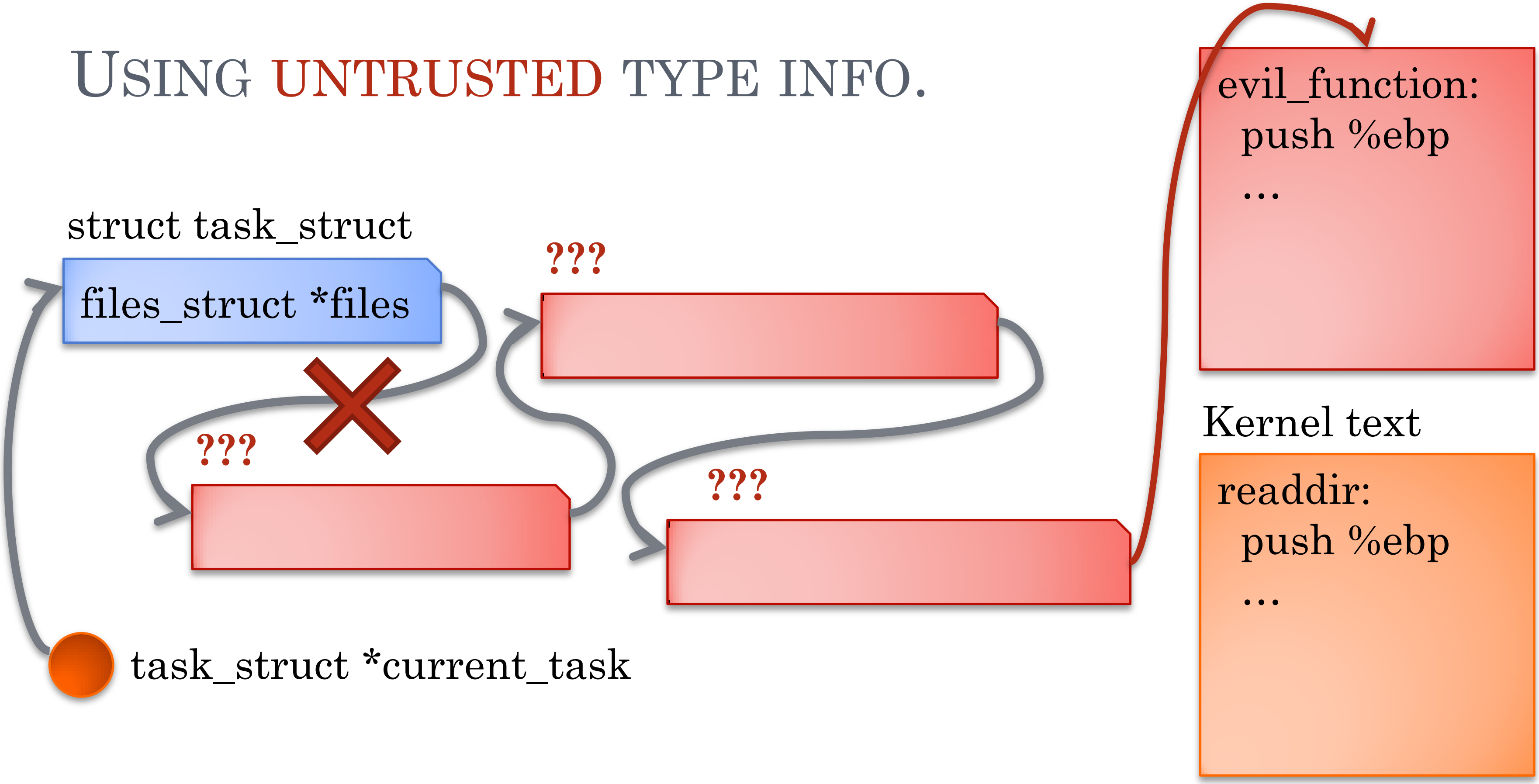
- Modify type information to mislead OSck?
  - Have to modify type information for predecessors

USING UNTRUSTED TYPE INFO.



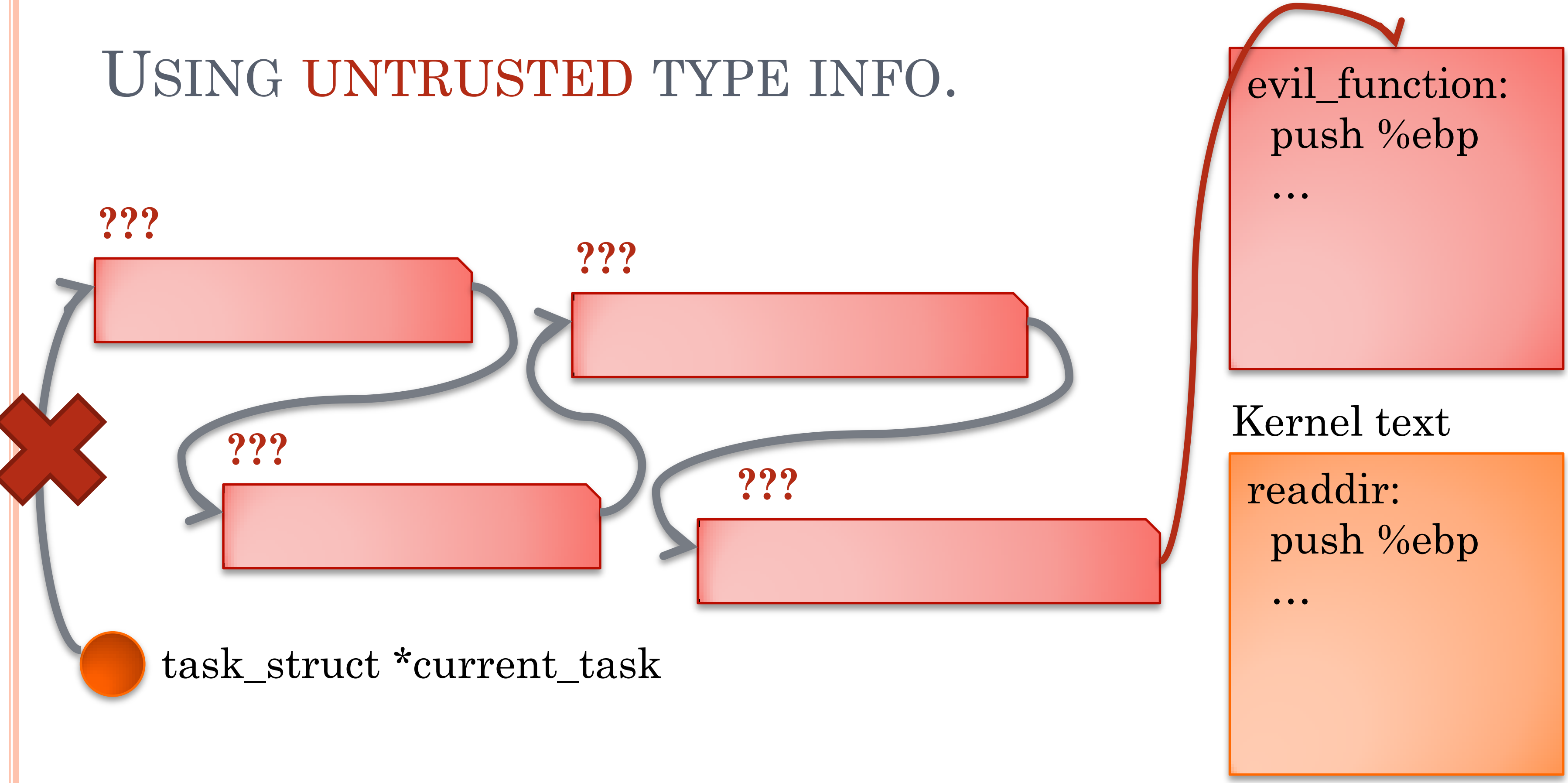
- Modify type information to mislead OSck?
  - Have to modify type information for predecessors

USING UNTRUSTED TYPE INFO.



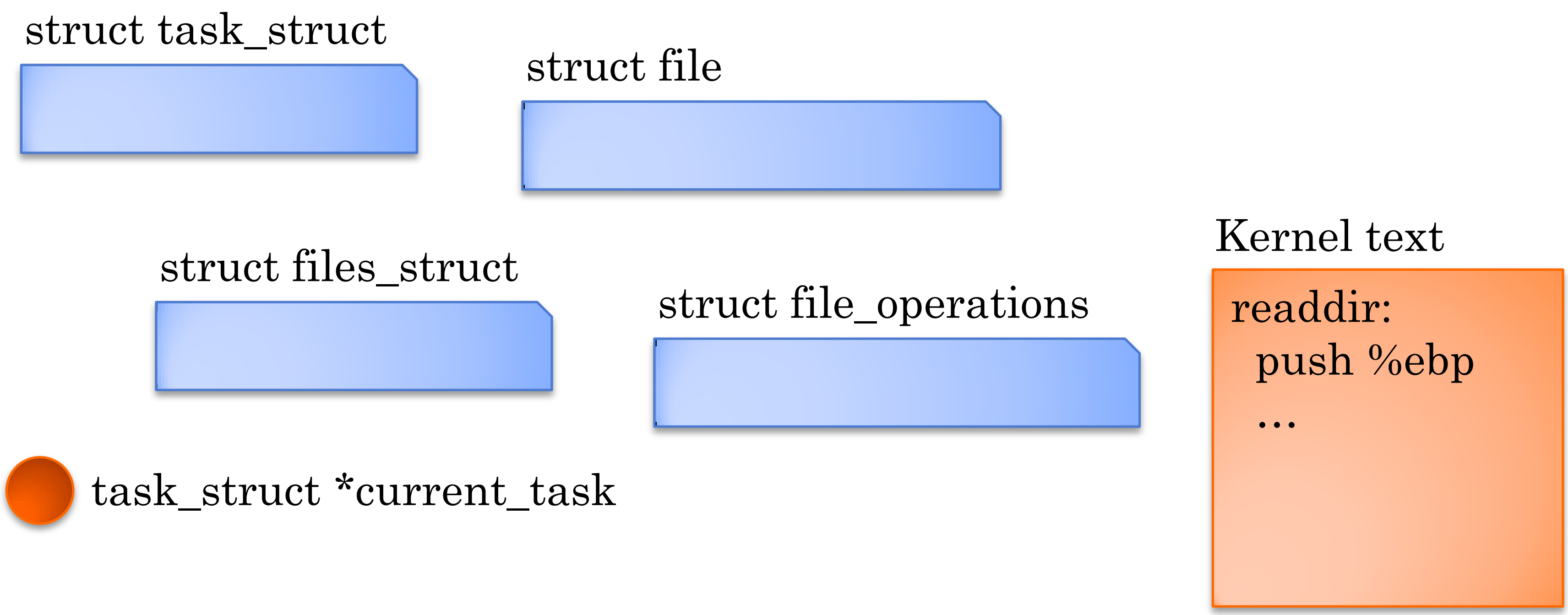
- Modify type information to mislead OSck?
  - Have to modify type information for predecessors

USING UNTRUSTED TYPE INFO.



- Cannot change type assigned to symbol
  - Compiled into kernel

USING UNTRUSTED TYPE INFO.



- Use type information for efficient checking
- Interpret type information from untrusted kernel
- Do not rely on type information for safety

OSck design goals

- Efficiency and safety
  - Verifier must inspect all kernel memory
  - Use hints from untrusted kernel to speed checks
- Programmability
  - Not all checks are automatic
  - Make it easy to write ad-hoc checks
  - Source-to-source translation of kernel data structures
- Concurrency
  - Checking code runs concurrently with kernel
  - Safely handle concurrency-related errors

Protecting non-control data

- Integrity for function pointers is well-specified through kernel source
  - Object *X* at offset *Y* points to *Z*

- Data integrity properties complicated, ad-hoc
  - e.g. list *A* == tree *B*
  - Can take a kernel developer's understanding

- Provide kernel-like interface for verifying properties
  - Extract data structure definitions
  - Source-to-source translation
  - Verification code looks like a kernel thread

Handling concurrency

- OSck runs concurrently with kernel execution
  - No synchronization with kernel
  - Data races possible
- Races can cause false negatives
  - Rootkit present, evades OSck with data race
  - Assume false negatives are not reproducible
- Races can cause false positives
  - Benign inconsistency causes OSck to detect rootkit
  - Adopt ‘stop the world’ approach

Evaluating design goals

- Efficiency and safety
  - How long do checks take to run?
  - What is the overhead on a running system?
  - What rootkits does OSck detect?

- Programmability
  - How much work is it to write data structure checks?

- Concurrency
  - How often does concurrency cause false positives?

How long do checks take?

Benchmark	Avg. time	Max time
SPEC INT 2006	76ms	123ms
RAB	109ms	316ms
Kernel compile	126ms	324ms

- Most system activity: ≈100ms
- Filesystem benchmarks have longer worst case
  - Create large numbers of kernel objects

What is the overhead?

	host	guest	OSck
SPEC 2006			
INT	1.00	1.03	+2%
FP	1.00	1.03	+0%
RAB			
mkdir	9.69	5.87	+2%
copy	35.6	44.07	+2%
du	0.23	0.39	+3%
grep/sum	3.37	1.89	-2%
Kernel compile			
	515	471	+0%

What rootkits does OSck detect?

- All of them
  - That we could find
- Take corpus of rootkits from available in the wild
  - Port some
  - Extract hiding vectors from others
  - Complete coverage of hiding vectors
- Develop new rootkit vectors
  - extable – corrupts exception table and pointers
  - ret-to-sched – creates hidden process by modifying stacks

How much work to detect rootkits?

- Function pointer type-safety most expansive property
  - 504 lines of C
- Other individual properties require little code
  - No individual check > 100 lines
- Total: 804 LOC

False positives from concurrency

- In benchmarking: none
  - Heavyweight handling okay
- Are they rare enough to be ignored?
  - High scheduling activity causes frequent updates to process list/tree
  - yield() microbenchmark causes false positives in 23% of scans

Conclusion

- OSck detects rootkits by verifying kernel invariants
- Efficient type-safety through cooperation with untrusted kernel
- Accessible interface for specifying ad-hoc data structure invariants
- Correct concurrency handling