CS 6431

Detecting and Preventing Memory Attacks

Vitaly Shmatikov

Intrusion Detection Techniques

- Misuse detection
  - Use attack “signatures” - need a model of the attack
  - Must know in advance what attacker will do (how?)
  - Can only detect known attacks
- Anomaly detection
  - Using a model of normal system behavior, try to detect deviations and abnormalities
  - Can potentially detect unknown attacks
- Which is harder to do?

Level of Monitoring

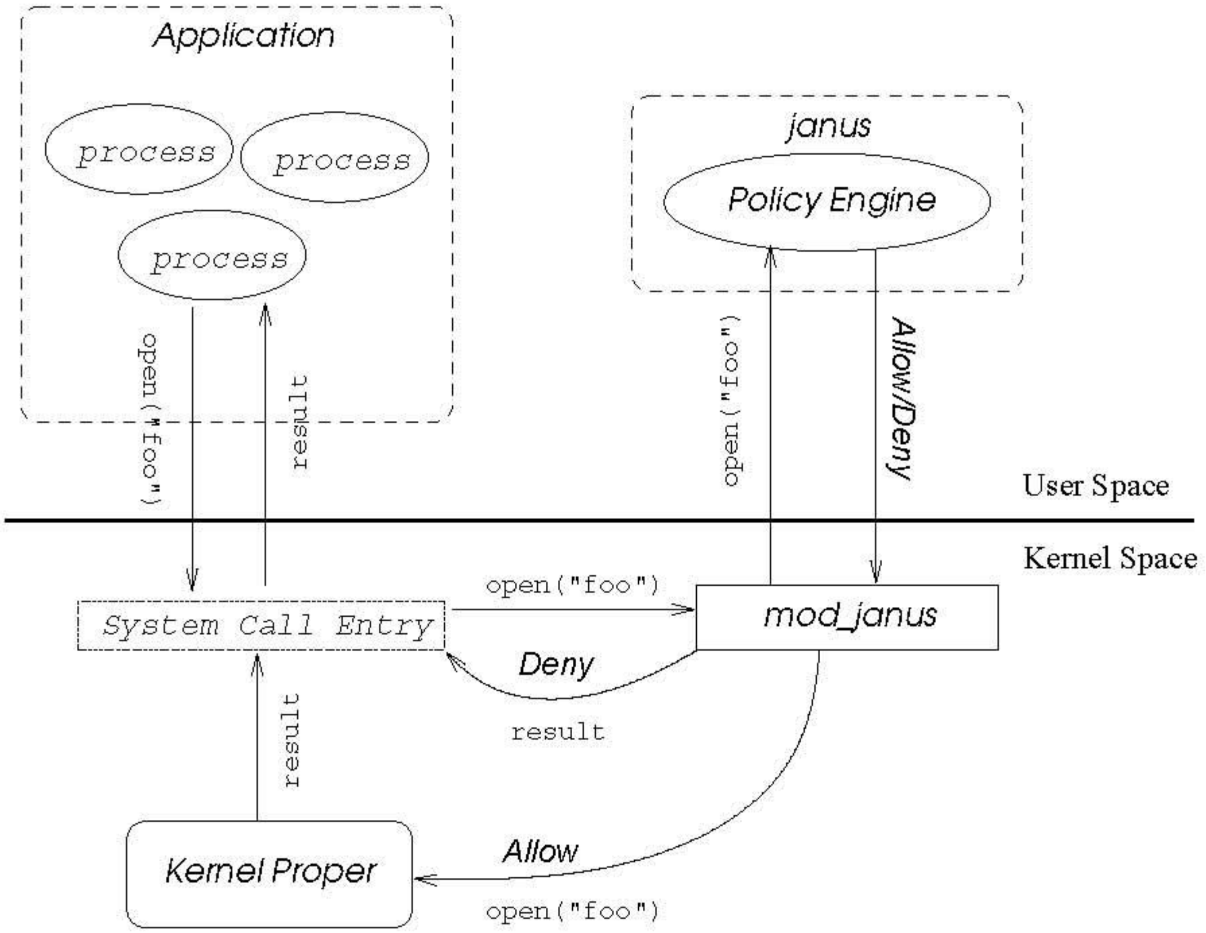
- Which types of events to monitor?
  - OS system calls
  - Command line
  - Network data (e.g., from routers and firewalls)
  - Keystrokes
  - File and device accesses
  - Memory accesses
- Auditing / monitoring should be scalable

System Call Interposition

- Observation: all sensitive system resources are accessed via OS system call interface
  - Files, sockets, etc.
- Idea: monitor all system calls and block those that violate security policy
  - Inline reference monitors
  - Language-level
    - Example: Java runtime environment inspects the stack of the function attempting to access a sensitive resource to check whether it is permitted to do so
  - Common OS-level approach: system call wrapper
    - Want to do this without modifying OS kernel (why?)

Janus

[Berkeley project, 1996]



Policy Design

- Designing a good system call policy is not easy
- When should a system call be permitted and when should it be denied?
- Example: ghostscript
  - Needs to open X windows
  - Needs to make X windows calls
  - But what if ghostscript reads characters you type in another X window?

Traps and Pitfalls

[Garfinkel. "Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools". NDSS 2003]

- Incorrectly mirroring OS state
- Overlooking indirect paths to resources
  - Inter-process sockets, core dumps
- Race conditions (TOCTTOU)
  - Symbolic links, relative paths, shared thread meta-data
- Unintended consequences of denying OS calls
  - Process dropped privileges using setuid but didn't check value returned by setuid… and monitor denied the call
- Bugs in reference monitors and safety checks
  - What if runtime environment has a buffer overflow?

Incorrectly Mirroring OS State

[Garfinkel]

Policy: “process can bind TCP sockets on port 80,
but cannot bind UDP sockets”

X = socket(UDP, …)	Monitor: “X is a UDP socket”
Y = socket(TCP, …)	Monitor: “Y is a TCP socket”
close(Y)	
dup2(X,Y)	Monitor’s state now inconsistent with OS
bind(Y, …)	Monitor: “Y is a TCP socket, Ok to bind”
	Oops!

TOCTTOU in Syscall Interposition

- User-level program makes a system call
  - Direct arguments in stack variables or registers
  - Indirect arguments are passed as pointers
- Wrapper enforces some security policy
  - Arguments are copied into kernel memory and analyzed and/or substituted by the syscall wrapper
- What if arguments change right here?
- If permitted by the wrapper, the call proceeds
  - Arguments are copied into kernel memory
  - Kernel executes the call

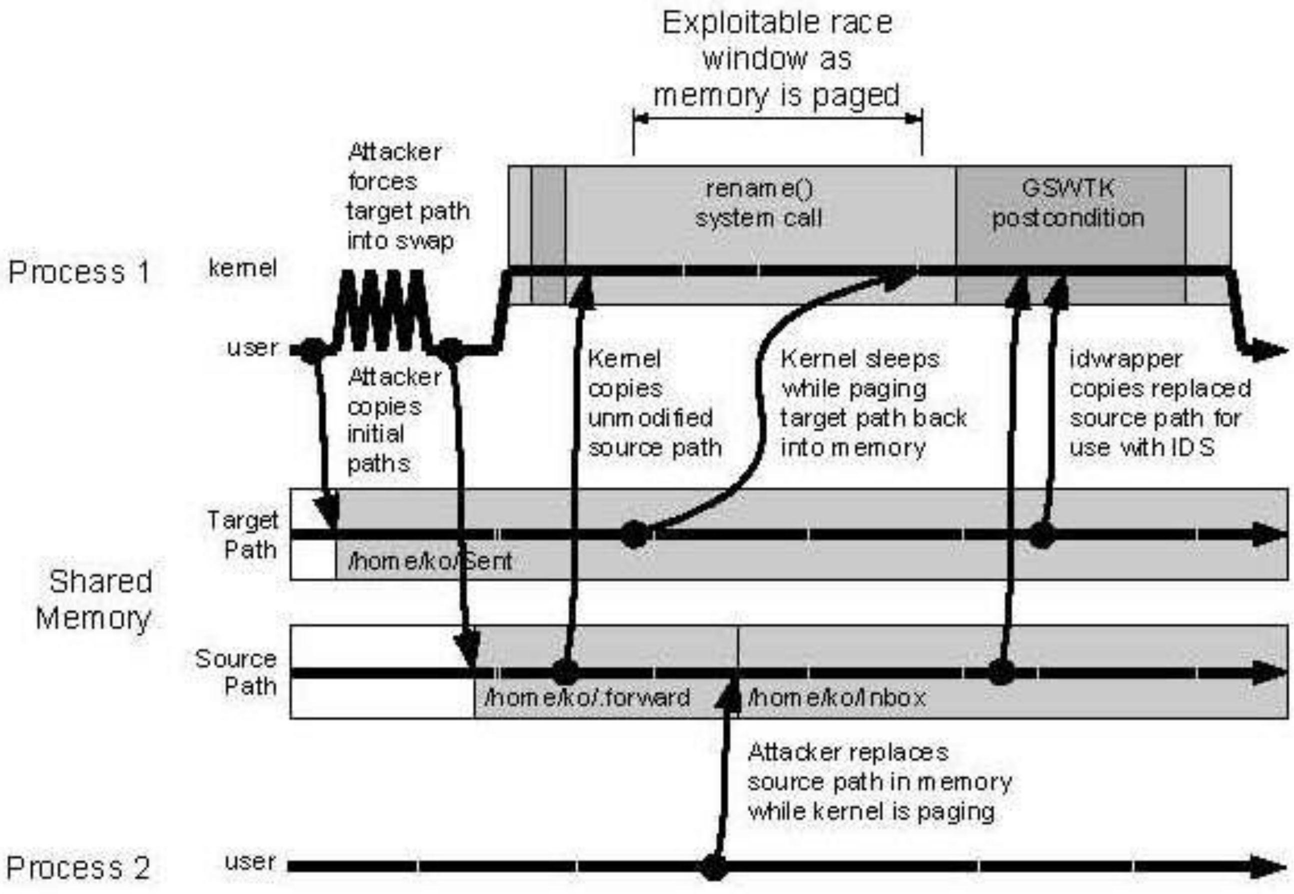
Exploiting TOCTTOU Conditions

[Watson. "Exploiting Concurrency Vulnerabilities in System Call Wrappers". WOOT 2007]

- Forced wait on disk I/O
  - Example: rename()
    - Attacker causes the target path of rename() to page out to disk
    - Kernel copies in the source path, then waits for target path
    - Concurrent attack process replaces the source path
    - Postcondition checker sees the replaced source path
- Voluntary thread sleeps
  - Example: TCP connect()
    - Kernel copies in the arguments
    - Thread calling connect() waits for a TCP ACK
    - Concurrent attack process replaces the arguments

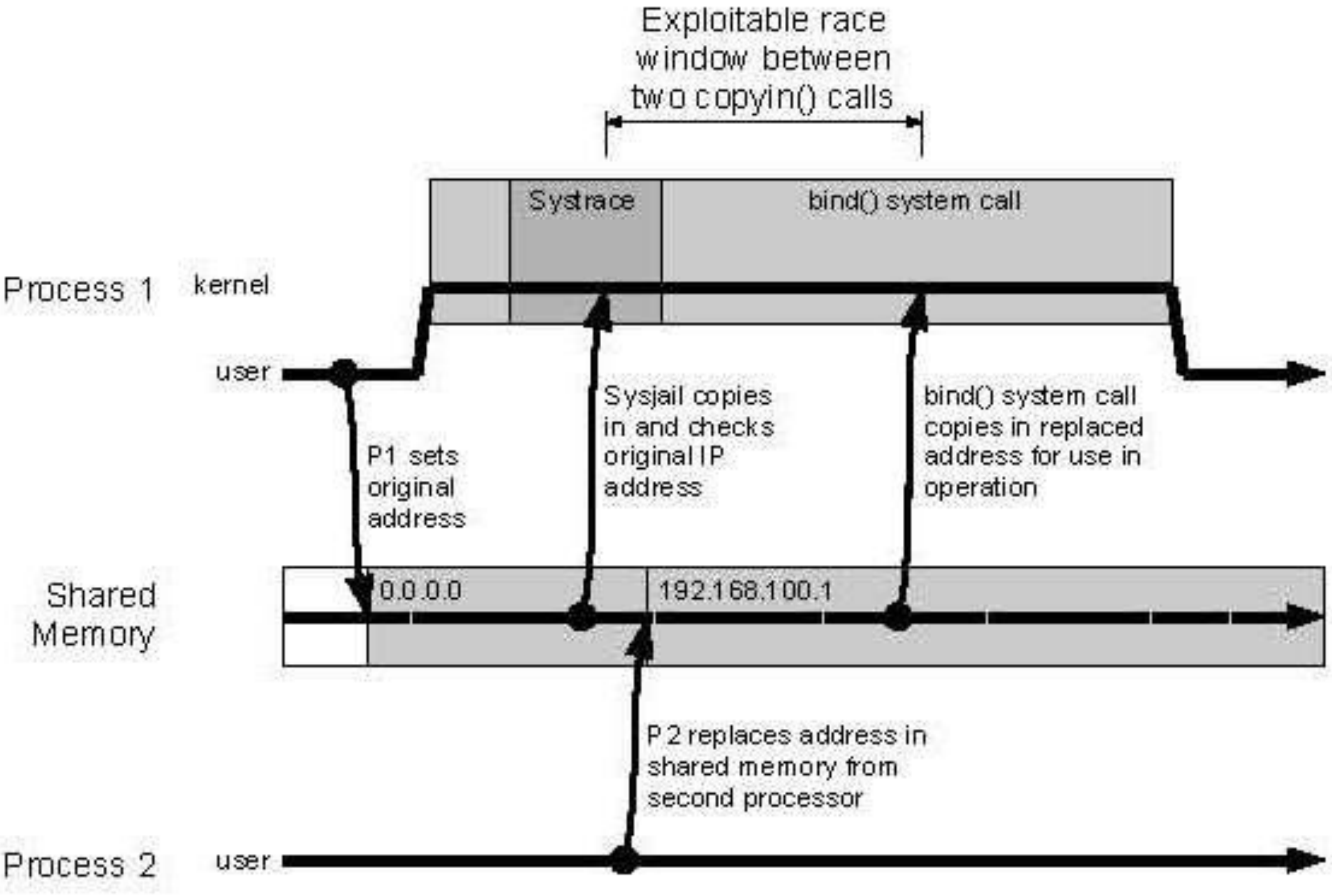
TOCTTOU via a Page Fault

[Watson]



TOCTTOU on Sysjail

[Watson]



Mitigating TOCTTOU

- Make pages with syscall arguments read-only
  - Tricky implementation issues
  - Prevents concurrent access to data on the same page
- Avoid shared memory between user process, syscall wrapper and the kernel
  - Argument caches used by both wrapper and kernel
  - Message passing instead of argument copying (why does this help?)
- Atomicity using system transactions
- Integrate security checks into the kernel?

Interposition + Static Analysis

1. Analyze the program to determine its expected system call behavior
2. Monitor actual behavior
3. Flag an intrusion if there is a deviation from the expected behavior
    - System call trace of the application is constrained to be consistent with the source or binary code
    - Main advantage: a conservative model of expected behavior will have zero false positives

Trivial "Bag-O'Calls" Model

- Determine the set S of all system calls that an application can potentially make
  - Lose all information about relative call order
- At runtime, check for each call whether it belongs to this set
- Problem: large number of false negatives
  - Attacker can use any system call from S
- Problem: |S| very big for large applications

Callgraph Model

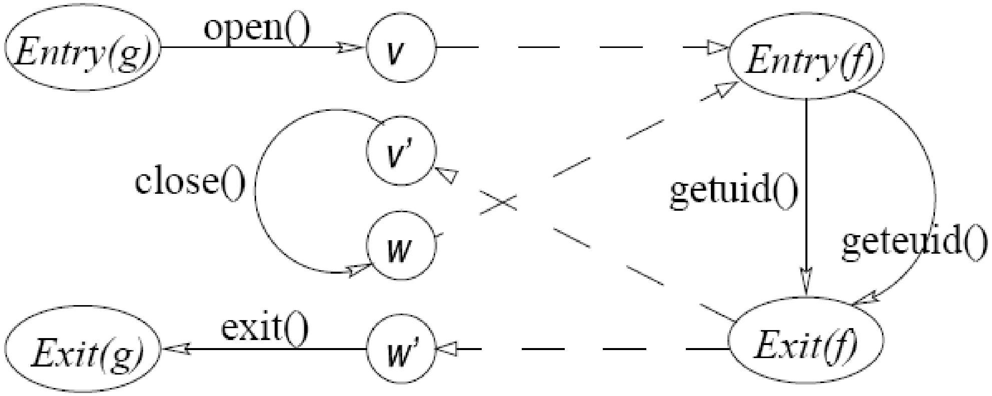
[Wagner and Dean. "Intrusion Detection via Static Analysis". Oakland 2001]

- Build a control-flow graph of the application by static analysis of its source or binary code
- Result: non-deterministic finite-state automaton (NFA) over the set of system calls
  - Each vertex executes at most one system call
  - Edges are system calls or empty transitions
  - Implicit transition to special "Wrong" state for all system calls other than the ones in original code; all other states are accepting
- System call automaton is conservative
  - Zero false positives!

NFA Example

[Wagner and Dean]

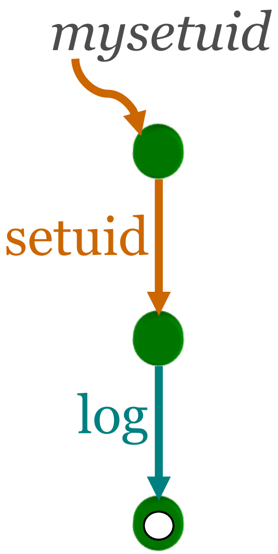
```
f(int x) {
  x ? getuid() : geteuid();
  x++;
}
g() {
  fd = open("foo", O_RDONLY);
  f(0); close(fd); f(1);
  exit(0);
}
```

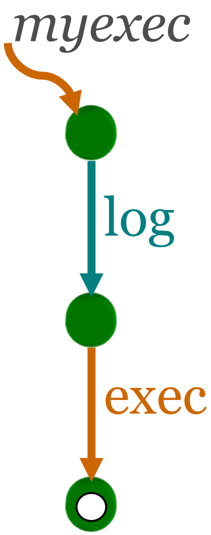


- Monitoring is O(|V|) per system call
- Problem: attacker can exploit impossible paths
  - The model has no information about stack state!

Another NFA Example

[Giffin]



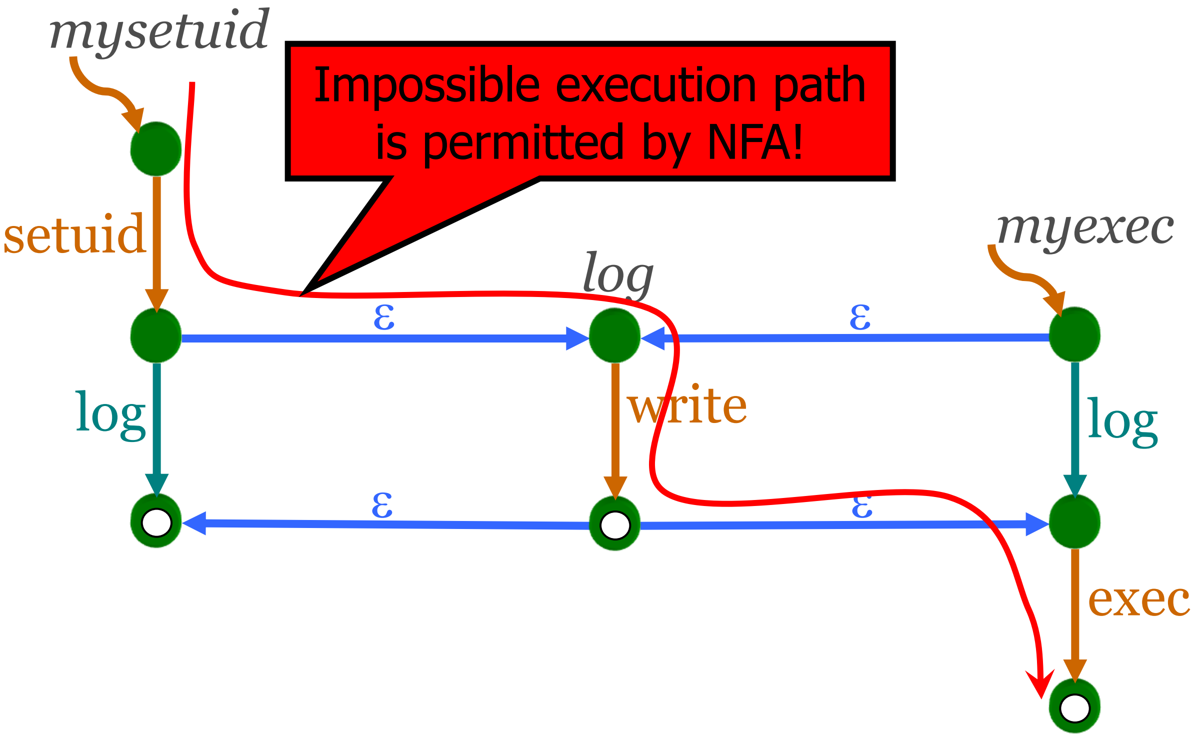


```
void
mysetuid (uid_t uid)
{
  setuid(uid);
  log("Set UID", 7);
}
```

```
void
log (char *msg,
     int len)
{
  write(fd, msg, len);
}
```

```
void
myexec (char *src)
{
  log("Execing", 7);
  exec("/bin/ls");
}
```

NFA Permits Impossible Paths



Impossible execution path is permitted by NFA!

NFA: Modeling Tradeoffs

- A good model should be…
  - Accurate: closely models expected execution
  - Fast: runtime verification is cheap

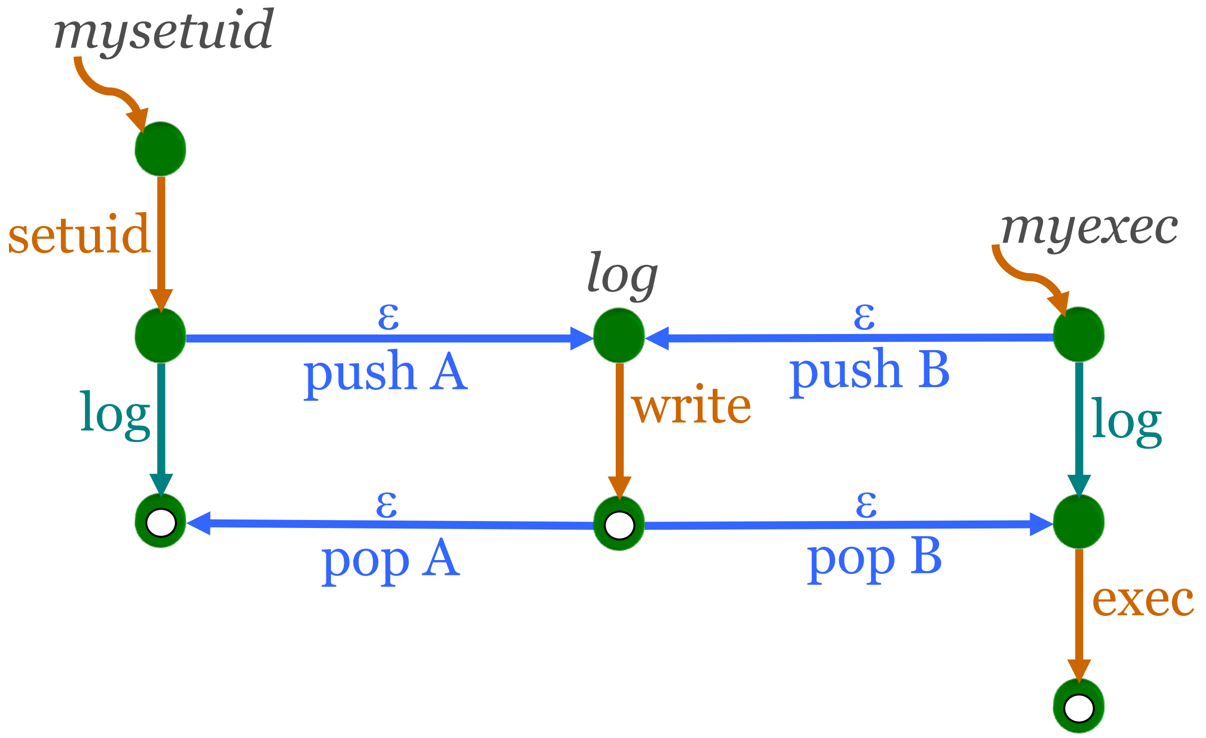
	Inaccurate	*Accurate*
Slow		
Fast	NFA	

Abstract Stack Model

- NFA is not precise, loses stack information
- Alternative: model application as a context-free language over the set of system calls
  - Build a non-deterministic pushdown automaton (PDA)
  - Each symbol on the PDA stack corresponds to single stack frame in the actual call stack
  - All valid call sequences accepted by PDA; enter "Wrong" state when an impossible call is made

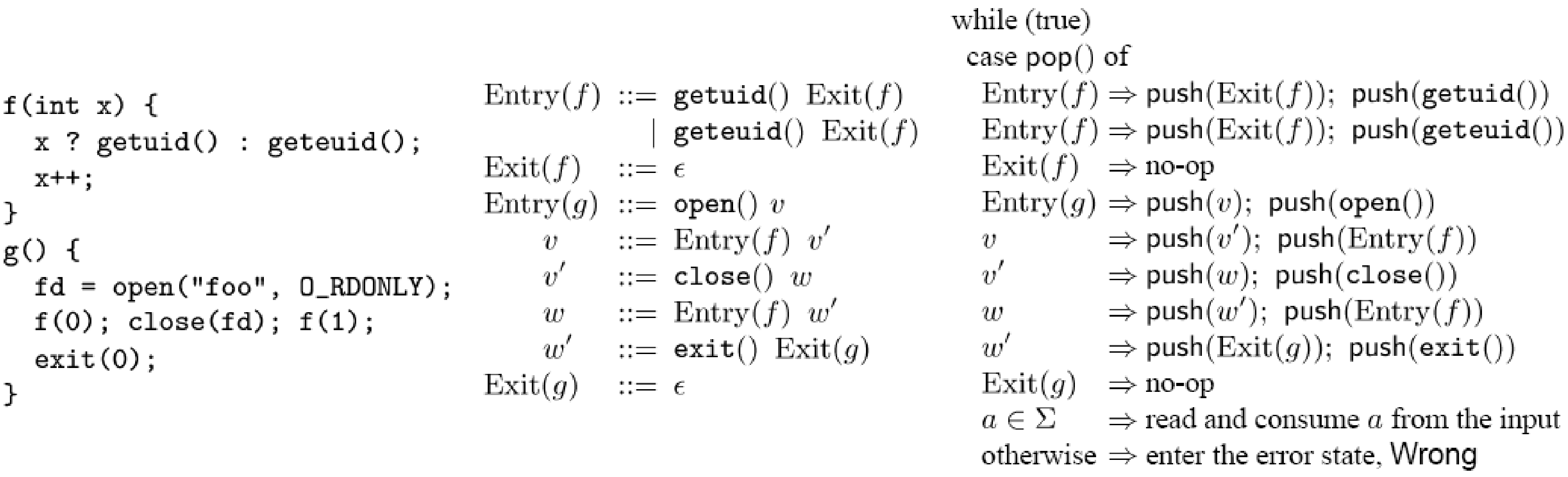
PDA Example

[Giffin]



Another PDA Example

[Wagner and Dean]



```
f(int x) {
  x ? getuid() : geteuid();
  x++;
}
g() {
  fd = open("foo", O_RDONLY);
  f(0); close(fd); f(1);
  exit(0);
}
```

$$\begin{array}{lcl}
\text{Entry}(f) & ::= & \texttt{getuid}()\ \text{Exit}(f) \\
 & | & \texttt{geteuid}()\ \text{Exit}(f) \\
\text{Exit}(f) & ::= & \epsilon \\
\text{Entry}(g) & ::= & \texttt{open}()\ v \\
v & ::= & \text{Entry}(f)\ v' \\
v' & ::= & \texttt{close}()\ w \\
w & ::= & \text{Entry}(f)\ w' \\
w' & ::= & \texttt{exit}()\ \text{Exit}(g) \\
\text{Exit}(g) & ::= & \epsilon
\end{array}$$

```
while (true)
  case pop() of
    $\text{Entry}(f) \Rightarrow$ push($\text{Exit}(f)$); push(getuid())
    $\text{Entry}(f) \Rightarrow$ push($\text{Exit}(f)$); push(geteuid())
    $\text{Exit}(f) \Rightarrow$ no-op
    $\text{Entry}(g) \Rightarrow$ push($v$); push(open())
    $v \Rightarrow$ push($v'$); push($\text{Entry}(f)$)
    $v' \Rightarrow$ push($w$); push(close())
    $w \Rightarrow$ push($w'$); push($\text{Entry}(f)$)
    $w' \Rightarrow$ push($\text{Exit}(g)$); push(exit())
    $\text{Exit}(g) \Rightarrow$ no-op
    $a \in \Sigma \Rightarrow$ read and consume $a$ from the input
    otherwise $\Rightarrow$ enter the error state, Wrong
```

PDA: Modeling Tradeoffs

- Non-deterministic PDA has high cost
  - Forward reachability algorithm is cubic in automaton size
  - Unusable for online checking

	Inaccurate	*Accurate*
Slow		PDA
Fast	NFA	

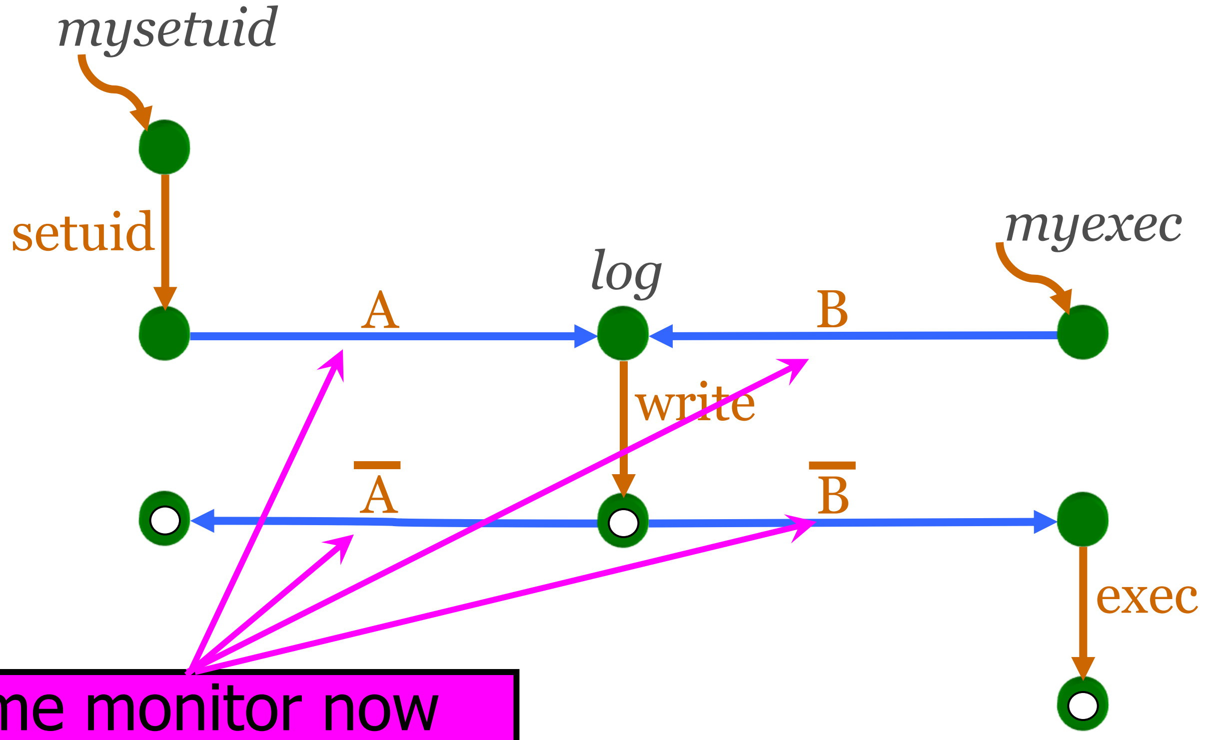
Dyck Model

[Giffin et al. "Efficient Context-Sensitive Intrusion Detection". NDSS 2004]

- Idea: make stack updates (i.e., function calls and returns) explicit symbols in the alphabet
  - Result: stack-deterministic PDA
- At each moment, the monitor knows where the monitored application is in its call stack
  - Only one valid stack configuration at any given time
- How does the monitor learn about function calls?
  - Use binary rewriting to instrument the code to issue special "null" system calls to notify the monitor
    - Potential high cost of introducing many new system calls
  - Can't rely on instrumentation if application is corrupted

Example of Dyck Model

[Giffin]



Runtime monitor now "sees" these transitions

System Call Processing Complexity

[Giffin]

Model	*Time & Space Complexity*
NFA	$O(n)$
PDA	$O(nm^2)$
Dyck	$O(n)$

n is state count

m is transition count

Runtime Bounds Checking

Referent object = buffer to which pointer points

- Actual size is available at runtime!

1. Modified pointer representation
   - Pointer keeps information about its referent object
   - Incompatible with external code, libraries, etc. ☹
2. Special table maps pointers to referent objects
   - Check referent object on every dereference
   - What if a pointer is modified by external code? ☹
3. Keep track of address range of each object
   - For every pointer arithmetic operation, check that the result points to the same referent object

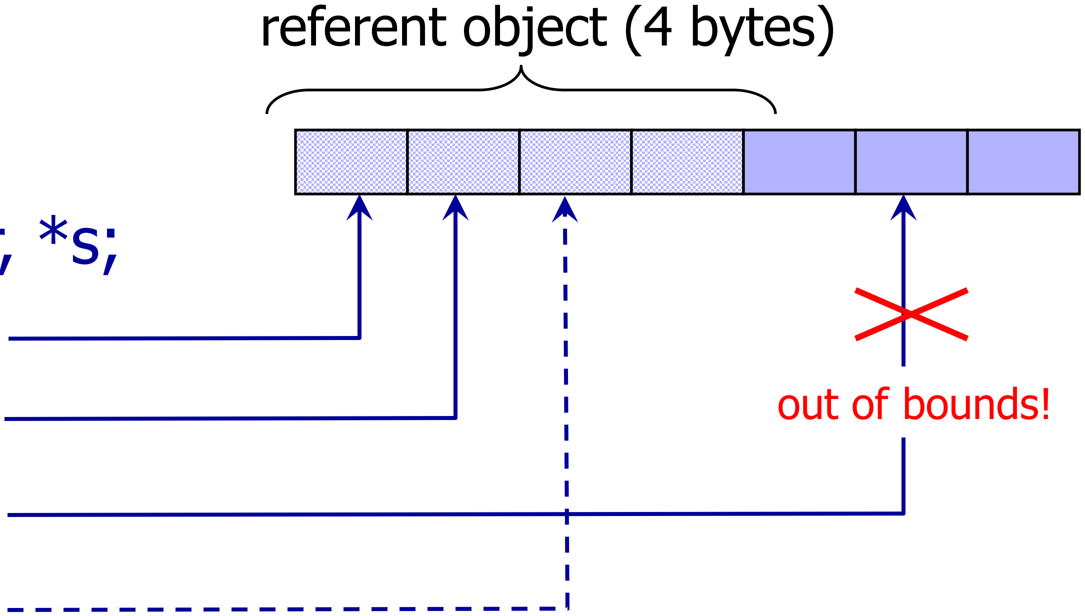
Jones-Kelly

[Jones and Kelly. "Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs". Automated and Algorithmic Debugging 1997]

- Pad each object by 1 byte
  - C permits a pointer to point to the byte right after an allocated memory object
- Maintain a runtime tree of allocated objects
- Backwards-compatible pointer representation
- Replace all out-of-bounds addresses with special ILLEGAL value (if dereferenced, program crashes)
- Problem: what if a pointer to an out-of-bounds address is used to compute an in-bounds address
  - Result: false alarm

Example of a False Alarm

```
{
  char *p, *q, *r, *s;
  p = malloc(4);
  q = p+1;
  s = p+5;
  r = s-3;
}
```



S is set to ILLEGAL

Program will crash if r is ever dereferenced

Note: this code works even though it's technically illegal in standard C

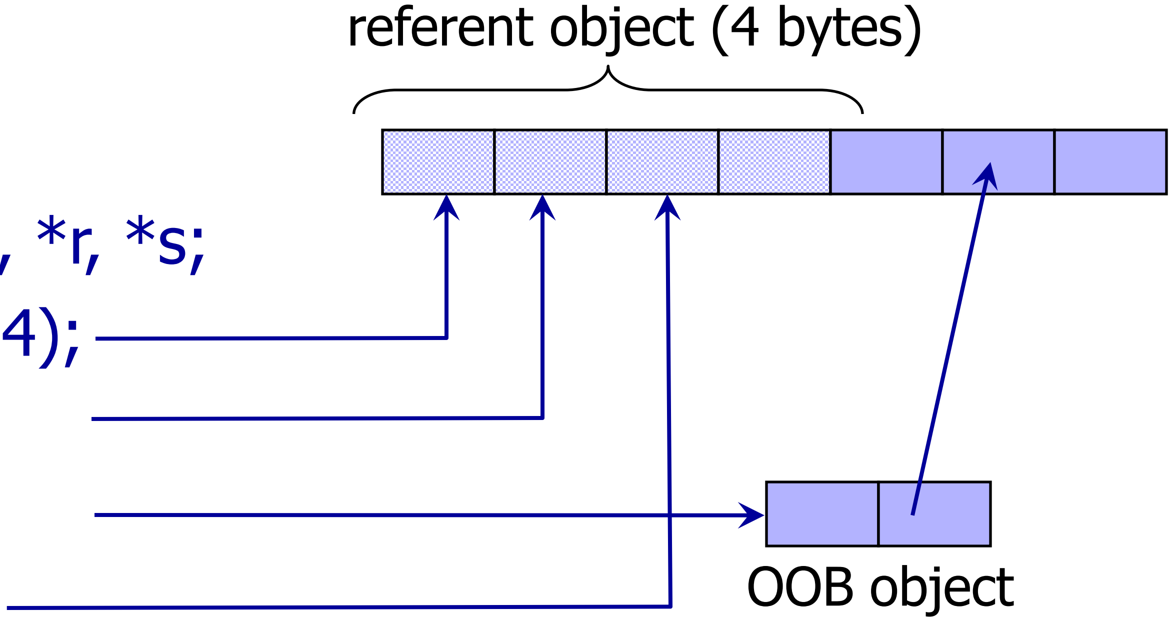
Ruwase-Lam

[Ruwase and Lam. "A Practical Dynamic Buffer Overflow Detector". NDSS 2004]

- Instead of ILLEGAL, make each out-of-bounds pointer point to a special OOB object
  - Stores the original out-of-bounds value
  - Stores a pointer to the original referent object
- Pointer arithmetic on out-of-bounds pointers
  - Simply use the actual value stored in the OOB object
- If a pointer is dereferenced, check if it points to an actual object. If not, halt the program!

Example of an OOB Object

```
{
  char *p, *q, *r, *s;
  p = malloc(4);
  q = p+1;
  s = p+5;
  r = s-3;
}
```



Value of r is in bounds

<u>Note</u>: this code works even though it’s technically illegal in standard C

Performance Issues

- Checking the referent object table on every pointer arithmetic operation is very expensive
- Jones-Kelly: 5x-6x slowdown
  - Tree of allocated objects grows very big
- Ruwase-Lam: 11x-12x slowdown if enforcing bounds on all objects, up to 2x if only strings
- Unusable in production code!

Dhurjati-Adve

[Dhurjati and Adve. "Backwards-compatible Array Bounds Checking for C with Very Low Overhead". ICSE 2006]

- Split memory into disjoint pools
  - Use aliasing information
  - Target pool for each pointer known at compile-time
  - Can check if allocation contains a single element (why does this help?)
- Separate tree of allocated objects for each pool
  - Smaller tree ⇒ much faster lookup; also caching
- Instead of returning a pointer to an OOB, return an address from the kernel address space
  - Separate table maps this address to the OOB
  - Don't need checks on every dereference (why?)

OOB Pointers: Ruwase-Lam

`p = malloc(10 * sizeof(int));` `q = p + 20;`	q = OOB(p+20,p) Put OOB(p+20,p) into a map
`r = q – 15;`	r = p + 5
`*r = … ; //no bounds overflow`	Check if r is out of bounds
`*q = … ; // overflow`	Check if q is out of bounds: Runtime error

Check on every dereference

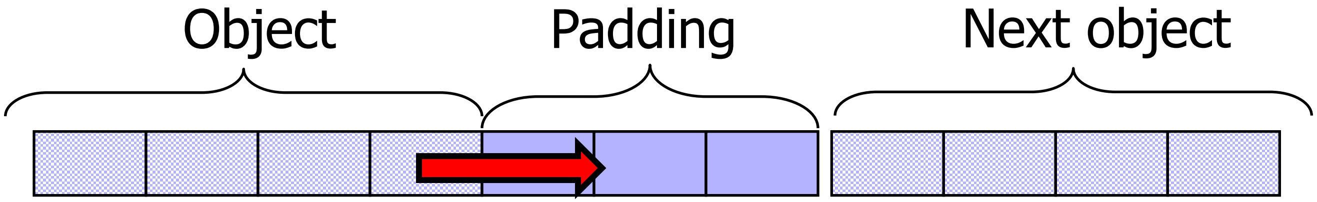
OOB Pointers: Dhurjati-Adve

p = malloc(10 * sizeof(int)); q = p + 20;	q = 0xCCCCCCCC Put (0xCCCCCCCC, OOB(p+20,p)) into a map
r = q – 15;	r = p + 5
*r = … ; //no bounds overflow	No software check necessary!
*q = … ; // overflow	No software check necessary! Runtime error

Baggy Bounds

[Akritidis et al. "Baggy Bounds Checking". USENIX Security 2009]

- Allocators pad objects to align pointers
- Insight: to prevent "bad" out-of-bounds memory accesses, it is enough to check allocation bounds, not the precise object bounds
  - What is a "bad" out-of-bounds access?



Out-of-bounds access to padding is harmless
(is this true?)

Very Efficient Bounds Representation

[Akritidis et al.]

- Storing the pointer to the object and its size requires at least 8 bytes per object
- Instead, use a custom allocator to pad and align objects to powers of 2 ⇒ then it’s enough to store log of object’s size in the bounds table

  e= log2(alloc_size) … this takes 1 byte per object and can be used to compute its size and base pointer:

  alloc_size = 1 << e

  base = p & ~(alloc_size-1)

Very Efficient Bounds Table

[Akritidis et al.]

◆ Partition memory into slots and align allocated objects to slot boundaries
  - Thus each slot can belong to at most 1 object

◆ Bounds table = contiguous array of 1-byte entries (an entry per each slot)

◆ Given an address p, finding its entry takes a single memory lookup
  - p>>log2(slot_size) + constant table base = address of the corresponding entry in the bounds table
  - No need for tree traversal!

Very Efficient Bounds Checking

[Akritidis et al.]

- Given a pointer arithmetic operation …

  p' = p + i

- … perform a very efficient check

  (p^p')>>BoundsTable[p>>log2(slot_size)] == 0

- This checks whether p and p' have the same prefix with only log(alloc_size) least significant bytes modified
  - No need to check against lower and upper bounds!

Handling Out-of-Bounds Pointers

[Akritidis et al.]

- Use a variant of Dhurjati-Adve technique to prevent OOB pointers from being dereferenced
  - Restrict the program to lower half of address space, set the most significant bit of OOB pointers to 1, thus all OOB look as if they point into kernel space
- Find the original referent object by checking whether the OOB pointer is in the top or bottom half of a slot (how does this work?)
  - Only works within slot_size/2 of the original object
  - On 64-bit architectures, can do better by using "spare" bits to tag each pointer with its bounds info

Baggy Bounds Check on x86

[Akritidis et al.]

```
mov eax, buf
shr eax, 4
mov al, byte ptr [TABLE+eax]
char *p = buf[i];
mov ebx, buf
xor ebx, p
shr ebx, al
jz ok
    p = slowPath(buf, p)
ok:
```

Assuming 16-byte slots

Look up log(object size)

Unsafe pointer operation

Check if p and buf differ in at most log(object size) least significant bytes

If buf is out of bounds, recover the referent object, remove OOB mark

Reference Monitor

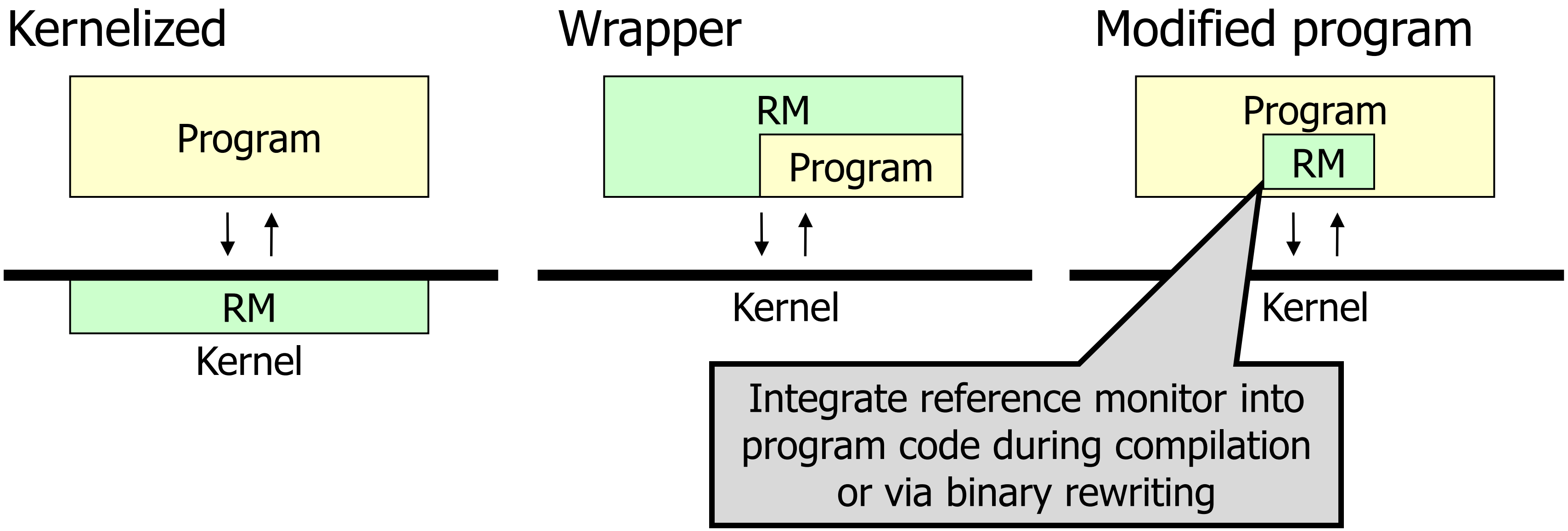
- Observes execution of the program/process
  - At what level? Possibilities: hardware, OS, network
- Halts or confines execution if the program is about to violate the security policy
  - What's a "security policy"?
  - Which system events are relevant to the policy?
    - Instructions, memory accesses, system calls, network packets…
- Cannot be circumvented by the monitored process

Enforceable Security Policies

[Schneider 1998]

- Reference monitors can only enforce safety policies
  - Execution of a process is a sequence of states
  - Safety policy is a predicate on a prefix of the sequence
    - Policy must depend only on the past of a particular execution; once it becomes false, it's always false
- Not policies that require knowledge of the future
  - "If this server accepts a SYN packet, it will eventually send a response"
- Not policies that deal with all possible executions
  - "This program should never reveal a secret"

Reference Monitor Implementation



Kernelized

Program

RM

Kernel

Wrapper

RM

Program

Kernel

Modified program

Program

RM

Kernel

Integrate reference monitor into program code during compilation or via binary rewriting

- Policies can depend on application semantics
- Enforcement doesn’t require context switches in the kernel
- Lower performance overhead

What Makes a Process Safe?

- Memory safety: all memory accesses are “correct”
  - Respect array bounds, don’t stomp on another process’s memory, don’t execute data as if it were code
- Control-flow safety: all control transfers are envisioned by the original program
  - No arbitrary jumps, no calls to library routines that the original program did not call
- Type safety: all function calls and operations have arguments of correct type

OS as a Reference Monitor

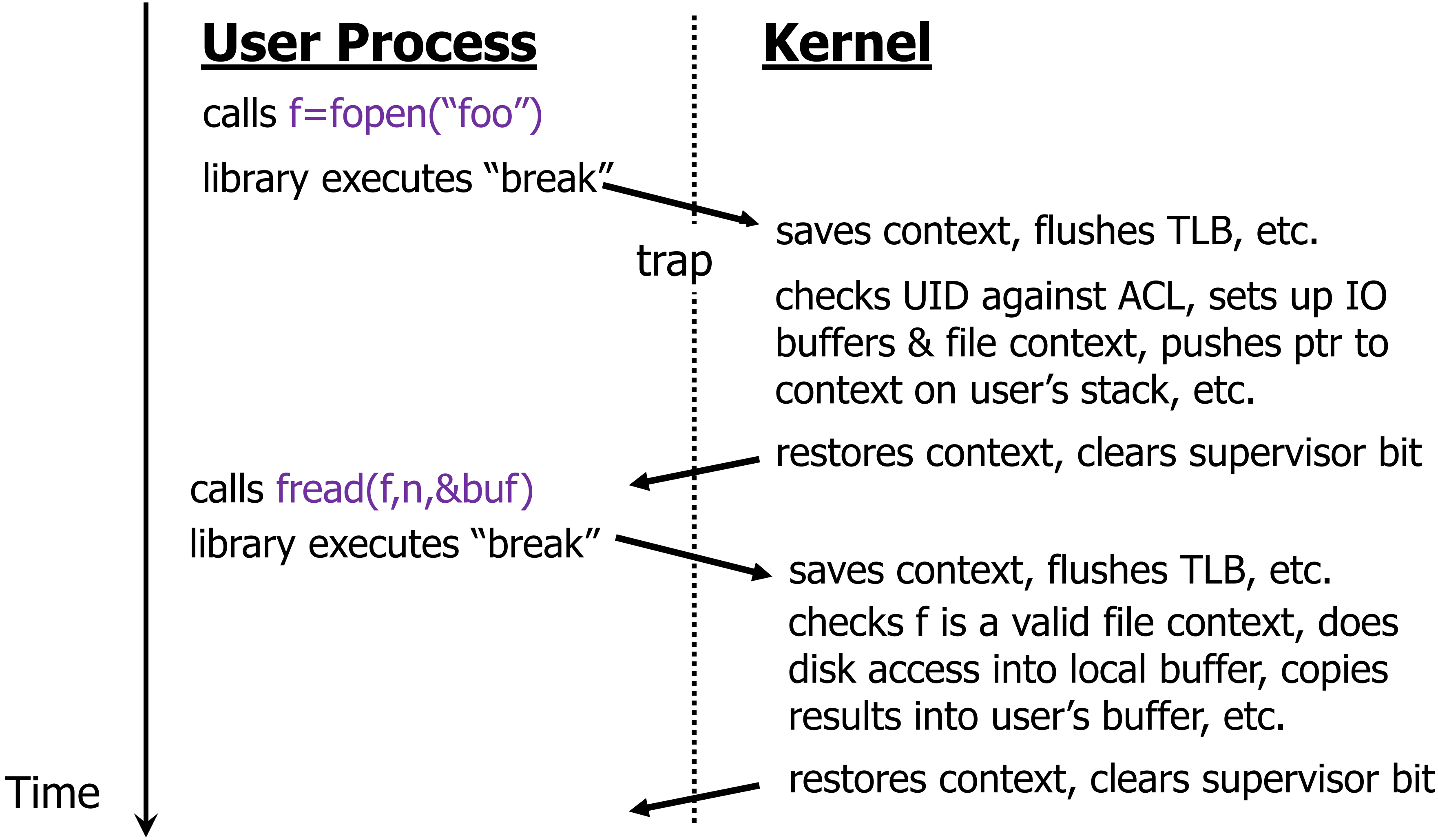
- Collection of running processes and files
  - Processes are associated with users
  - Files have access control lists (ACLs) saying which users can read/write/execute them
- OS enforces a variety of safety policies
  - File accesses are checked against file's ACL
  - Process cannot write into memory of another process
  - Some operations require superuser privileges
    - But may need to switch back and forth (e.g., setuid in Unix)
  - Enforce CPU sharing, disk quotas, etc.
- Same policy for all processes of the same user

Hardware Mechanisms: TLB

- TLB: Translation Lookaside Buffer
  - Maps virtual to physical addresses
  - Located next to the cache
  - Only supervisor process can manipulate TLB
    - But if OS is compromised, malicious code can abuse TLB to make itself invisible in virtual memory (Shadow Walker)
- TLB miss raises a page fault exception
  - Control is transferred to OS (in supervisor mode)
  - OS brings the missing page to the memory
- This is an expensive context switch

Steps in a System Call

[Morrisett]



User Process

Kernel

calls f=fopen("foo")

library executes "break"

trap

saves context, flushes TLB, etc.

checks UID against ACL, sets up IO buffers & file context, pushes ptr to context on user's stack, etc.

restores context, clears supervisor bit

calls fread(f,n,&buf)

library executes "break"

saves context, flushes TLB, etc.

checks f is a valid file context, does disk access into local buffer, copies results into user's buffer, etc.

restores context, clears supervisor bit

Time

Modern Hardware Meets Security

- Modern hardware: large number of registers, big memory pages
- Isolation ⇒ each process should live in its own hardware address space
- … but the performance cost of inter-process communication is increasing
  - Context switches are very expensive
  - Trapping into OS kernel requires flushing TLB and cache, computing jump destination, copying memory
- Conflict: isolation vs. cheap communication

Software Fault Isolation (SFI)

[Wahbe et al. SOSP 1993]

- Processes live in the same hardware address space; software reference monitor isolates them
  - Each process is assigned a logical “fault domain”
  - Check all memory references and jumps to ensure they don’t leave process’s domain
- Tradeoff: checking vs. communication
  - Pay the cost of executing checks for each memory write and control transfer to save the cost of context switching when trapping into the kernel

Fault Domains

- Process's code and data in one memory segment
  - Identified by a unique pattern of upper bits
  - Code is separate from data (heap, stack, etc.)
  - Think of a fault domain as a "sandbox"
- Binary modified so that it cannot escape domain
  - Addresses are masked so that all memory writes are to addresses within the segment
    - Coarse-grained memory safety (vs. array bounds checking)
  - Code is inserted before each jump to ensure that the destination is within the segment
- Does this help much against buffer overflows?

Verifying Jumps and Stores

- If target address can be determined statically, mask it with the segment's upper bits
  - Crash, but won't stomp on another process's memory
- If address unknown until runtime, insert checking code before the instruction
- Ensure that code can't jump around the checks
  - Target address held in a dedicated register
  - Its value is changed only by inserted code, atomically, and only with a value from the data segment

Simple SFI Example

- Fault domain = from 0x1200 to 0x12FF
- Original code: `write x`
- Naïve SFI:

```
x := x & 00FF
x := x | 1200
```

convert x into an address that lies within the fault domain

```
...
write x
```

What if the code jumps right here?

- Better SFI:

```
tmp := x & 00FF
tmp := tmp | 1200
write tmp
```

Inline Reference Monitor

- Generalize SFI to more general safety policies than just memory safety
  - Policy specified in some formal language
  - Policy deals with application-level concepts: access to system resources, network events, etc.
    - "No process should send to the network after reading a file", "No process should open more than 3 windows", …
- Policy checks are integrated into the binary code
  - Via binary rewriting or when compiling
- Inserted checks should be uncircumventable
  - Rely on SFI for basic memory safety

CFI: Control-Flow Integrity

[Abadi et al. "Control-Flow Integrity". CCS 2005]

- Main idea: pre-determine control flow graph (CFG) of an application
  - Static analysis of source code
  - Static binary analysis ← CFI
  - Execution profiling
  - Explicit specification of security policy
- Execution must follow the pre-determined control flow graph

CFI: Binary Instrumentation

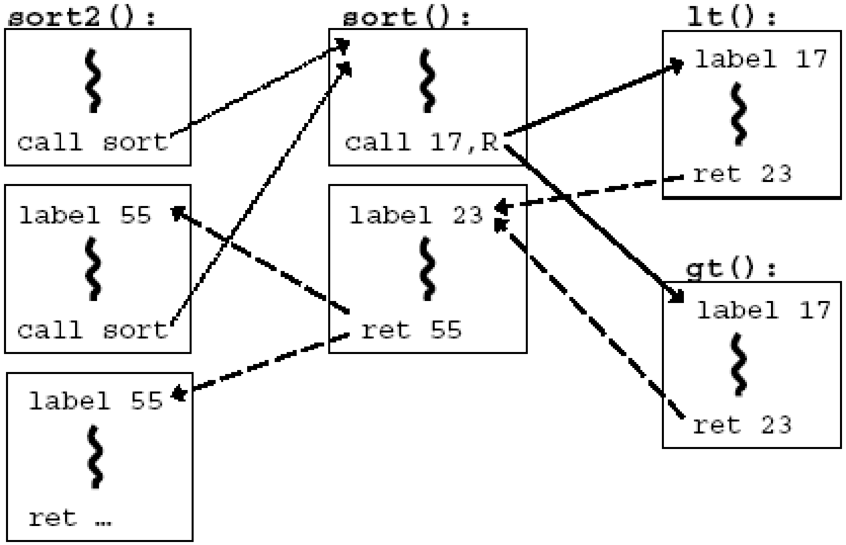
- Use binary rewriting to instrument code with runtime checks (similar to SFI)
- Inserted checks ensure that the execution always stays within the statically determined CFG
  - Whenever an instruction transfers control, destination must be valid according to the CFG
- Goal: prevent injection of arbitrary code and invalid control transfers (e.g., return-to-libc)
  - Secure even if the attacker has complete control over the thread’s address space

CFG Example

```
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```

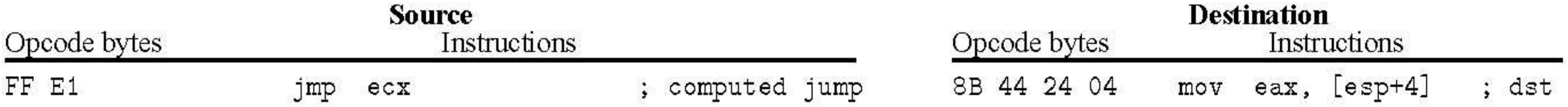


CFI: Control Flow Enforcement

- For each control transfer, determine statically its possible destination(s)
- Insert a unique bit pattern at every destination
  - Two destinations are equivalent if CFG contains edges to each from the same source
    - This is imprecise (why?)
  - Use same bit pattern for equivalent destinations
- Insert binary code that at runtime will check whether the bit pattern of the target instruction matches the pattern of possible destinations

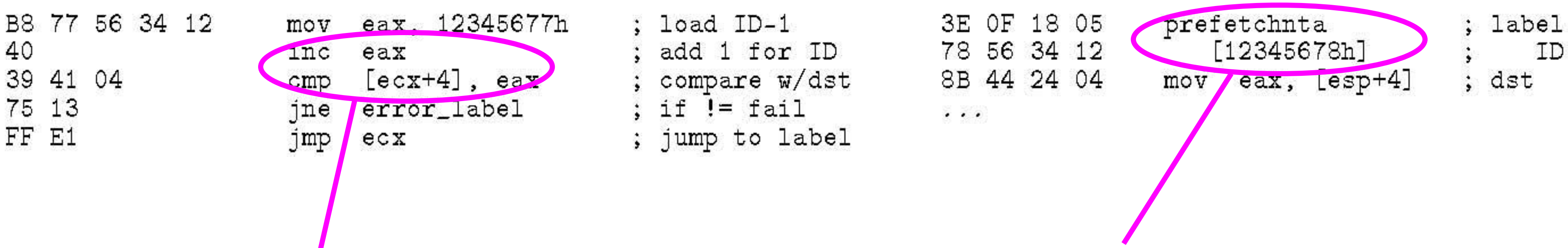
CFI: Example of Instrumentation

Original code



```
                          Source                                                     Destination
Opcode bytes                  Instructions                          Opcode bytes           Instructions
------------------------------------------------------------        -------------------------------------------
FF E1              jmp   ecx                ; computed jump         8B 44 24 04    mov   eax, [esp+4]    ; dst
```

Instrumented code



```
B8 77 56 34 12     mov   eax, 12345677h    ; load ID-1          3E 0F 18 05    prefetchnta         ; label
40                 inc   eax               ; add 1 for ID       78 56 34 12       [12345678h]      ;    ID
39 41 04           cmp   [ecx+4], eax      ; compare w/dst      8B 44 24 04    mov   eax, [esp+4]  ; dst
75 13              jne   error_label       ; if != fail         ...
FF E1              jmp   ecx               ; jump to label
```

Jump to the destination only if the tag is equal to "12345678"

Abuse an x86 assembly instruction to insert "12345678" tag into the binary

CFI: Preventing Circumvention

- Unique IDs
  - Bit patterns chosen as destination IDs must not appear anywhere else in the code memory except ID checks
- Non-writable code
  - Program should not modify code memory at runtime
    - What about run-time code generation and self-modification?
- Non-executable data
  - Program should not execute data as if it were code
- Enforcement: hardware support + prohibit system calls that change protection state + verification at load-time

Improving CFI Precision

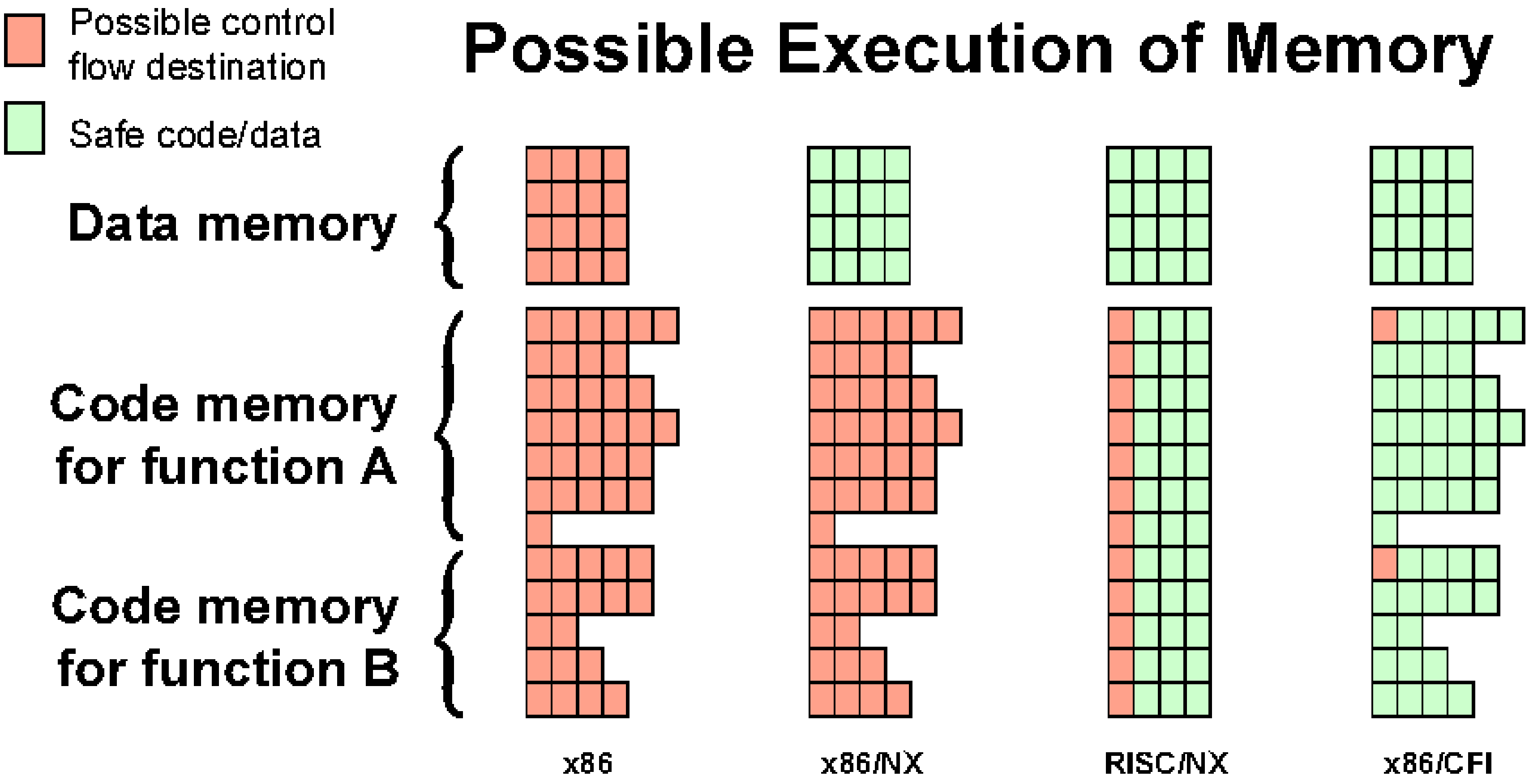
- Suppose a call from A goes to C, and a call from B goes to either C, or D (when can this happen?)
  - CFI will use the same tag for C and D, but this allows an "invalid" call from A to D
  - Possible solution: duplicate code or inline
  - Possible solution: multiple tags
- Function F is called first from A, then from B; what's a valid destination for its return?
  - CFI will use the same tag for both call sites, but this allows F to return to B after being called from A
  - Solution: shadow call stack

CFI: Security Guarantees

- Effective against attacks based on illegitimate control-flow transfer
  - Stack-based buffer overflow, return-to-libc exploits, pointer subterfuge
- Does not protect against attacks that do not violate the program’s original CFG
  - Incorrect arguments to system calls
  - Substitution of file names
  - Other data-only attacks

Possible Execution of Memory

[Erlingsson]



WIT: Write Integrity Testing

[Akritidis et al. "Preventing Memory Error Exploits with WIT". Oakland 2008]

◆ Combines static analysis …
- For each memory write, compute the set of memory locations that may be the destination of the write
- For each indirect control transfer, compute the set of addresses that may be the destination of the transfer
- "Color table" assigns matching colors to instruction (write or jump) and all statically valid destinations
  - Is this sound? Complete?

◆ … with dynamic enforcement
- Code is instrumented with runtime checks to verify that destination of write or jump has the right color

WIT: Write Safety Analysis

- Start with off-the-shelf “points-to” analysis
  - Gives a conservative set of possible values for each ptr
- A memory write instruction is “safe” if…
  - It has no explicit destination operand, or destination operand is a temporary, local or global variable
    - Such instructions either modify registers, or a constant number of bytes starting at a constant offset from the frame pointer or the data segment (example?)
  - … or writes through a pointer that is always in bounds
    - How do we know statically that a pointer is always in bounds?
- Safe instructions require no runtime checks
- Can also infer safe destinations (how?)

WIT: Runtime Checks

- Statically, assign a distinct color to each <u>un</u>safe write instruction and all of its possible destinations
  - What if some destination can be written by two different instructions? Any security implications?
- Add a runtime check that destination color matches the statically assigned color
  - What attack is this intended to prevent?
- Same for indirect (computed) control transfers
  - Except for indirect jumps to library functions (done through pointers which are protected by write safety)
  - How is this different from CFI? Hint: think RET address

WIT: Additional Protections

- Change layout of stack frames to segregate safe and unsafe local variables
- Surround unsafe objects by guards/canaries
  - What attack is this intended to prevent? How?
- Wrappers for malloc()/calloc() and free()
  - malloc() assigns color to newly allocated memory
  - free() is complicated
    - Has the same, statically computed color as the freed object
    - At runtime, treated as an unsafe write to this object
    - Reset color of object to 0 (what attack does this prevent?)
    - Several other subtle details and checks

WIT: Handling Libraries

- Basic WIT doesn't work for libraries (why?)
- Instead, assign the same, standard color to all unsafe objects allocated by library functions and surround them by guards
  - Different from the colors of safe objects and guards
  - What attack does this not prevent?
- Wrappers for memory copying functions
  - For example, memcpy() and strcpy()
  - Receive color of the destination as an extra argument, check at runtime that it matches static color

Native Client

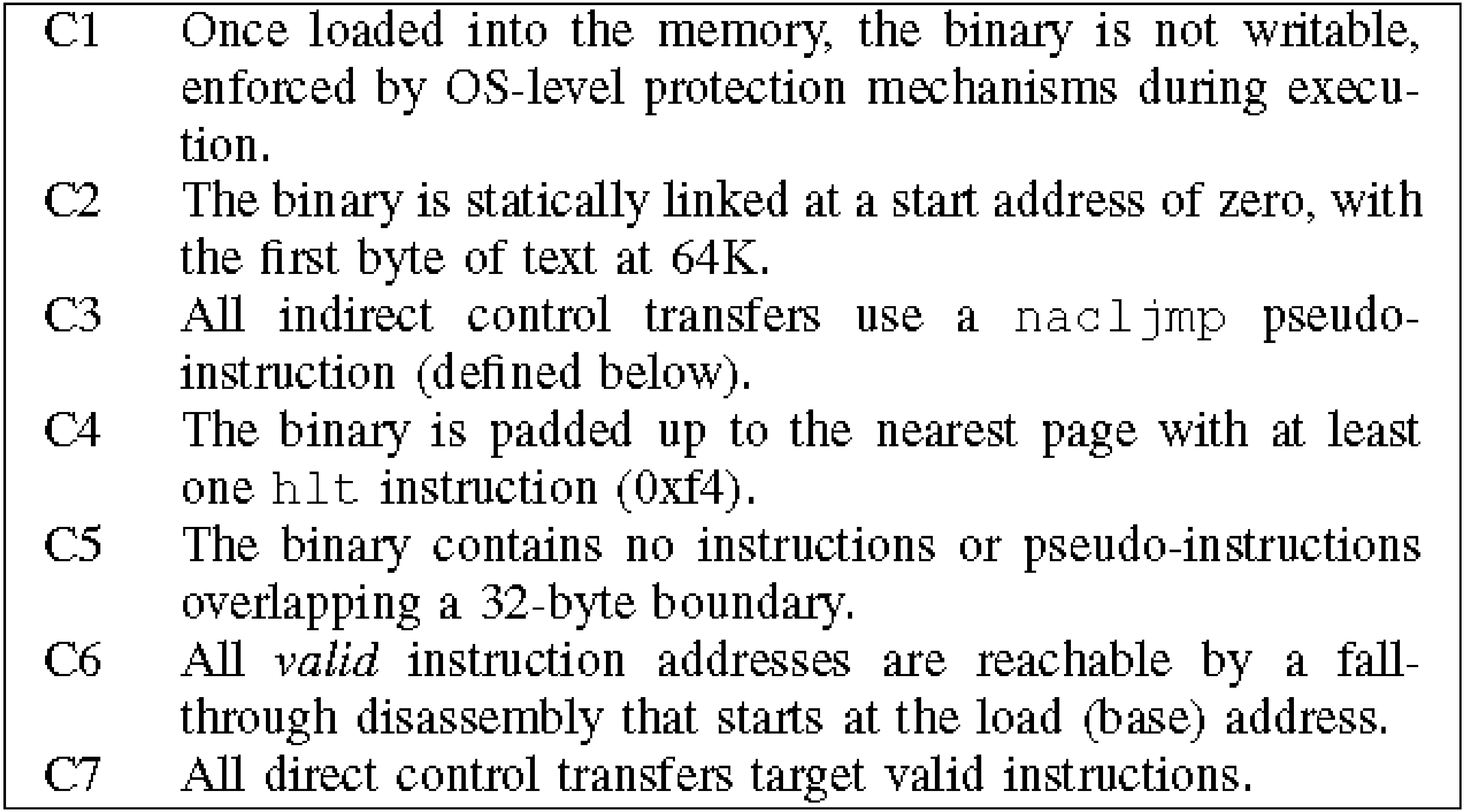
[Yee et al. "Native Client". Oakland 2009]

- Goal: download an x86 binary and run it "safely"
  - Much better performance than JavaScript, Java, etc.
- ActiveX: verify signature, then unrestricted
  - Critically depends on user's understanding of trust
- .NET controls: IL bytecode + verification
- Native Client: sandbox for untrusted x86 code
  - Restricted subset of x86 assembly
  - SFI-like sandbox ensures memory safety
  - Restricted system interface
  - (Close to) native performance

NaCl Sandbox

- Code is restricted to a subset of x86 assembly
  - Enables reliable disassembly and efficient validation
  - No unsafe instructions
    - syscall, int, ret, memory-dependent jmp and call, privileged instructions, modifications of segment state …
- No loads or stores outside dedicated segment
  - Address space constrained to 0 mod 32 segment
  - Similar to SFI
- Control-flow integrity

Constraints for NaCl Binaries



C1	Once loaded into the memory, the binary is not writable, enforced by OS-level protection mechanisms during execution.
C2	The binary is statically linked at a start address of zero, with the first byte of text at 64K.
C3	All indirect control transfers use a `nacljmp` pseudo-instruction (defined below).
C4	The binary is padded up to the nearest page with at least one `hlt` instruction (0xf4).
C5	The binary contains no instructions or pseudo-instructions overlapping a 32-byte boundary.
C6	All *valid* instruction addresses are reachable by a fall-through disassembly that starts at the load (base) address.
C7	All direct control transfers target valid instructions.

Control-Flow Integrity in NaCl

- For each direct branch, statically compute target and verify that it's a valid instruction
  - Must be reachable by fall-through disassembly
- Indirect branches must be encoded as

```
and  %eax, 0xffffffe0
jmp  *%eax
```

  - Guarantees that target is 32-byte aligned
  - Works because of restriction to the zero-based segment
  - Very efficient enforcement of control-flow integrity
- No RET
  - Sandboxing sequence, then indirect jump

Interacting with Host Machine

- Trusted runtime environment for thread creation, memory management, other system services
- Untrusted → trusted control transfer: trampolines
  - Start at 0 mod 32 addresses (why?) in the first 64K of the NaCl module address space
    - First 4K are read- and write-protected (why?)
  - Reset registers, restore thread stack (outside module’s address space), invoke trusted service handlers
- Trusted → untrusted control transfer: springboard
  - Start at non-0 mod 32 addresses (why?)
  - Can jump to any untrusted address, start threads

Other Aspects of NaCl Sandbox

- No hardware exceptions or external interrupts
  - Because segment register is used for isolation, stack appears invalid to the OS ⇒ no way to handle
- No network access via OS, only via JavaScript in browser
  - No system calls such as connect() and accept()
  - JavaScript networking is subject to same-origin policy
- IMC: inter-module communication service
  - Special IPC socket-like abstraction
  - Accessible from JavaScript via DOM object, can be passed around and used to establish shared memory