Software Foundations of Security and Privacy (15-316, spring 2017)

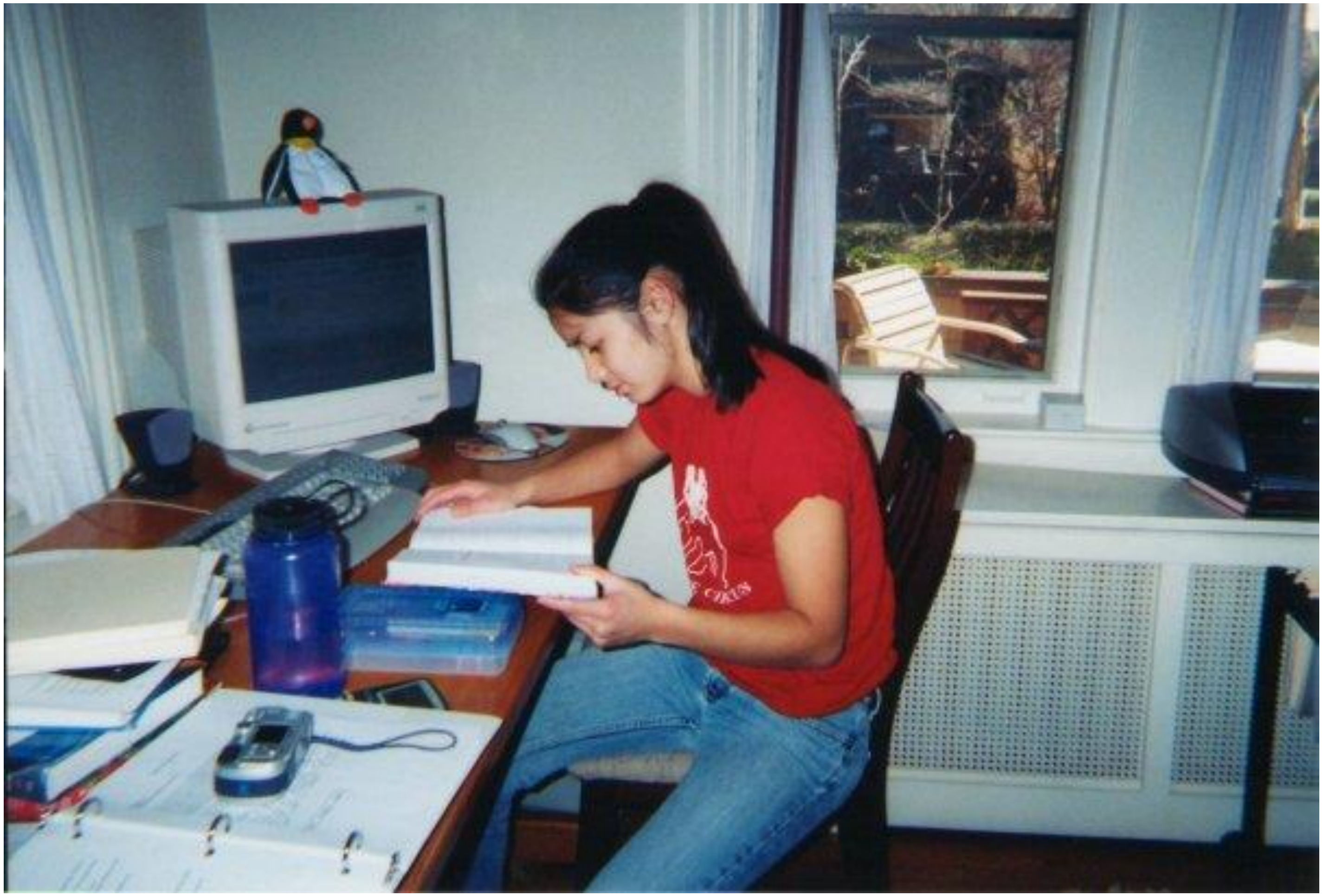
Lecture 6: Inline Reference Monitors

Jean Yang

jyang2@andrew.cmu.edu

With material from Vitaly Shmatikov and Ulfar Urlingsson

A tale of monitoring…



How this relates to our class

- **Desired safety property:** Jean never goes on AOL Instant Messenger.
- **Enforcement mechanism:** Mom checks Internet Explorer cookies. (This is an **audit**-based security mechanism!)
- **What Mom would like:** a mechanism for preventing Jean from going on AIM *before* it happens.

Solution: reference monitors!

Leftovers from Lecture 5

Mechanism integrity

- To correctly enforce a policy, we must assume:
  - Input symbols correspond to actual execution.
  - Transitions correspond to automaton’s true transition function.
- If target corrects mechanism, it can violate these assumptions.
- Address with two strategies:
  - **Isolation:** target must be unable to write to internal representation of automaton.
  - **Complete mediation:** make sure all aspects of execution that might generate input symbols are covered by implementation.

Proving correct enforcement

Goal: Show that when *S* executes under enforcement of SA *P*:

- *S* terminates when its execution violates *P*
- *S* continues to execute otherwise

This requires a proof that the implementation satisfies:

1. Complete mediation.
2. Target control.
3. Isolation.

Later, we will see how different implementation strategies lead to different kinds of proof!

More pragmatics

Two mechanisms are needed to implement SA:

- **Input read:** Determines that an input symbol has been produced by the target and forwards that symbol to the automaton simulation.
- **Transition:** Determines whether the automaton can make a transition on a given input symbol, and if so, executes that transition by updating automaton state effectively.

These implementations affect correctness and performance!

Part One: The Reference Monitor Framework

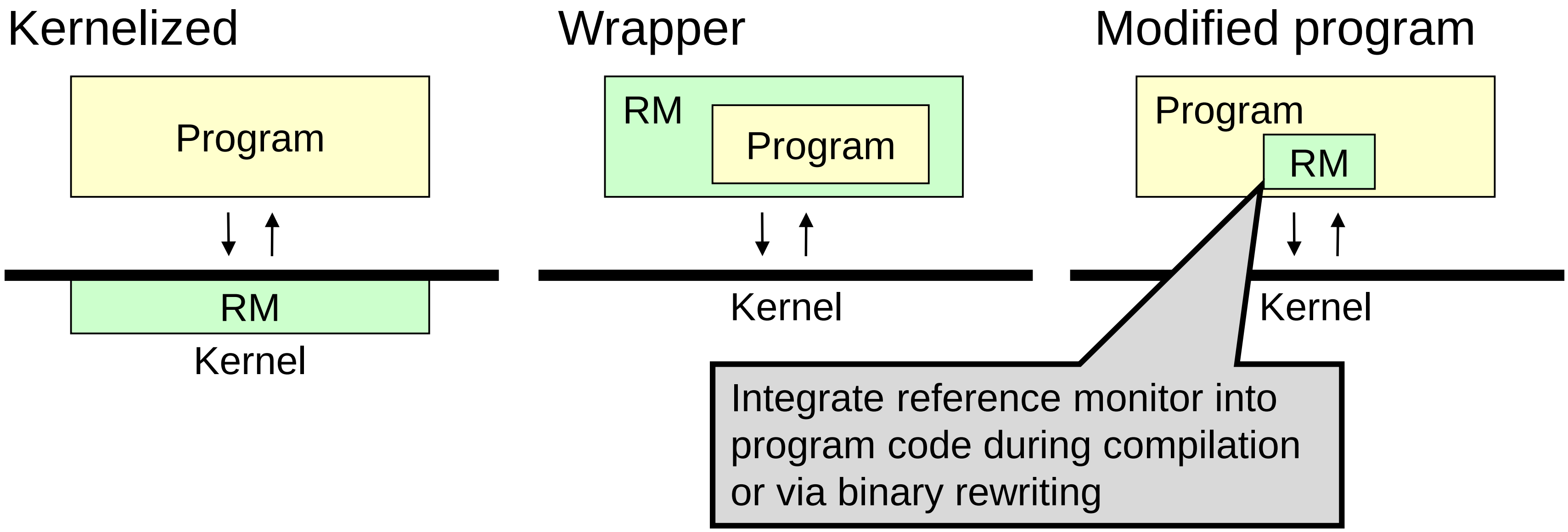
Recap: reference monitors

Slide source: Vitaly Shmatikov

- Observes execution of the program/process
  - Possible abstraction levels: hardware, OS, network
- Halts or contain 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
- Most enforcement mechanisms we will see are example of reference monitors

Reference monitor implementations

Slide source: Vitaly Shmatikov



- Policies can depend on application semantics
- Enforcement doesn't require context switches in the kernel

OS as a reference monitor

Slide source: Vitaly Shmatikov

- 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

Validity checks

Slide source: Ulfar Erlingsson

- Triggered by reference monitor on each event
- Encode the security policy
- Perform arbitrary computation to decide whether to allow event or halt
  - Can have side effects? (Not if EM!)
  - Can change program flow? (Not if EM!)

My work: how can we enforce safety properties if we want to have side effects and change the program flow?

Inline reference monitors

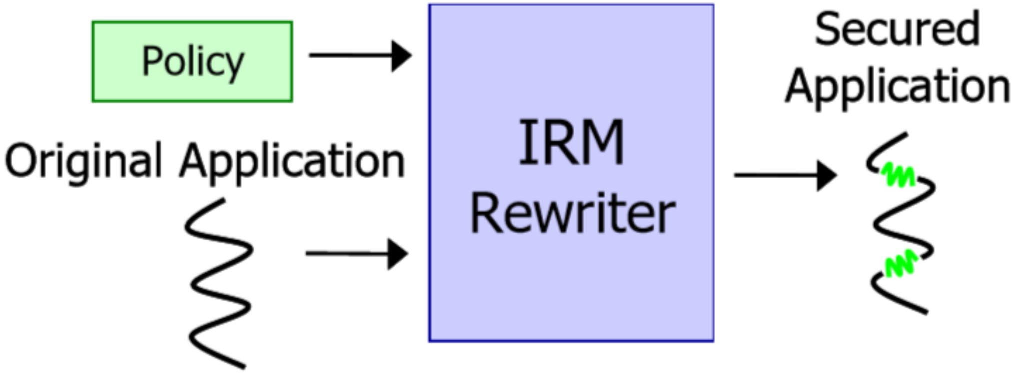
Slide source: Vitaly Shmatikov

- 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

Implementing IRMs by program modification

[Erlingsson Schneider 99]

Slide source: Ulfar Erlingsson

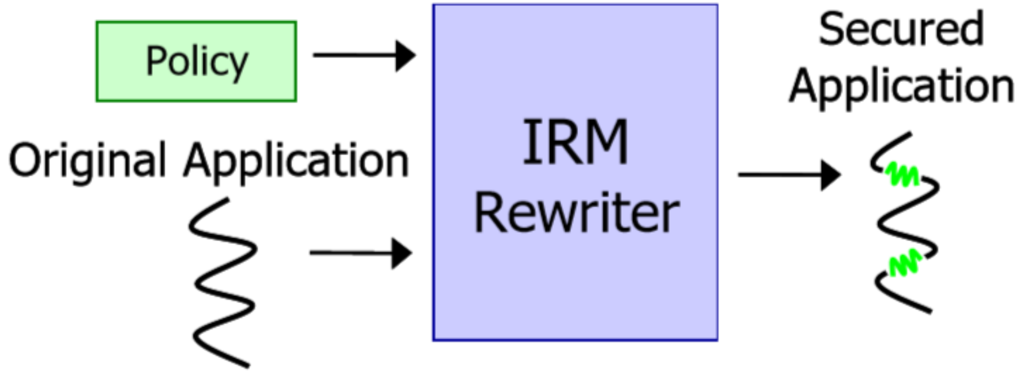


- Have access to program abstractions to capture all potential security-relevant events.
- Rewriter works on machine language programs.

Challenges in implementing IRMs

[Erlingsson Schneider 99]

Slide source: Ulfar Erlingsson



- How to capture all relevant events?
- Prevent application from subverting reference monitor
- Preserve application behavior

IRM enforcement advantages

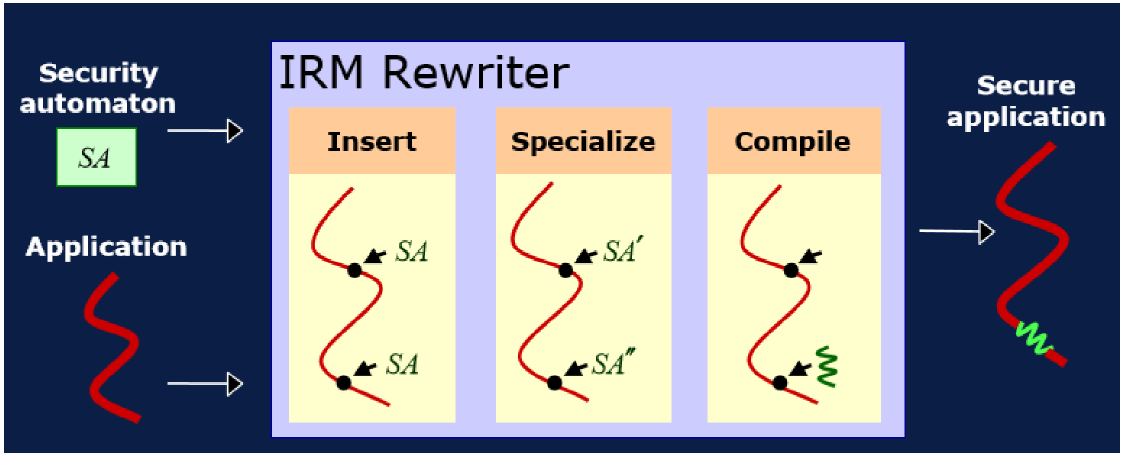
Slide source: Ulfar Erlingsson

- Can enforce policies on application abstractions (for instance MSWord macros and documents)
- Each application can have a distinct policy
  - Enforcement overhead determined by policy
  - Mechanism customized to policy
- Mechanism is simple and efficient

Efficient IRM enforcement

Slide source: Ulfar Erlingsson

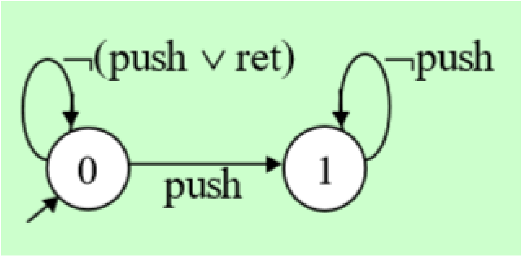
- Evaluate security automata policy at every point in the program
- Simplify security automata by partial evaluation using static knowledge

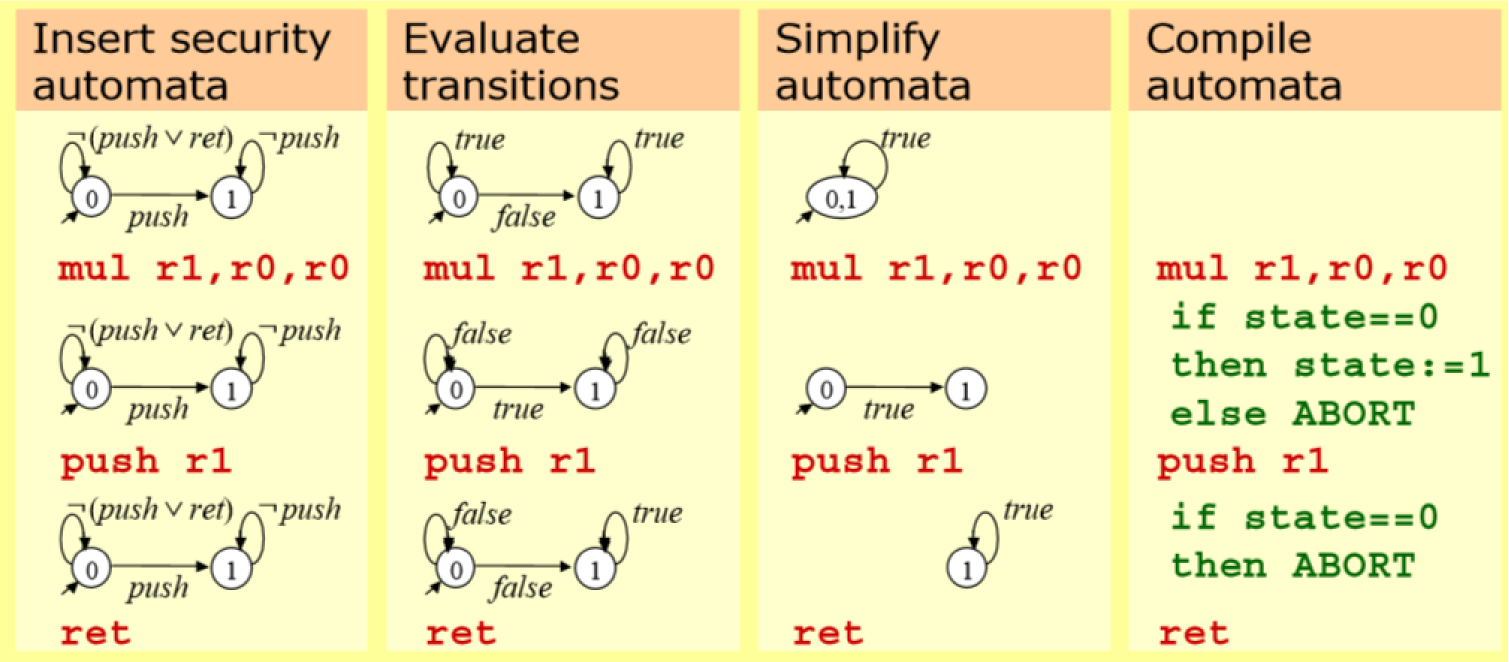


Example IRM rewriting

Slide source: Ulfar Erlingsson

Policy: push exactly once before returning





Part Two: From Policies to Reference Monitors

Enforceable security policies

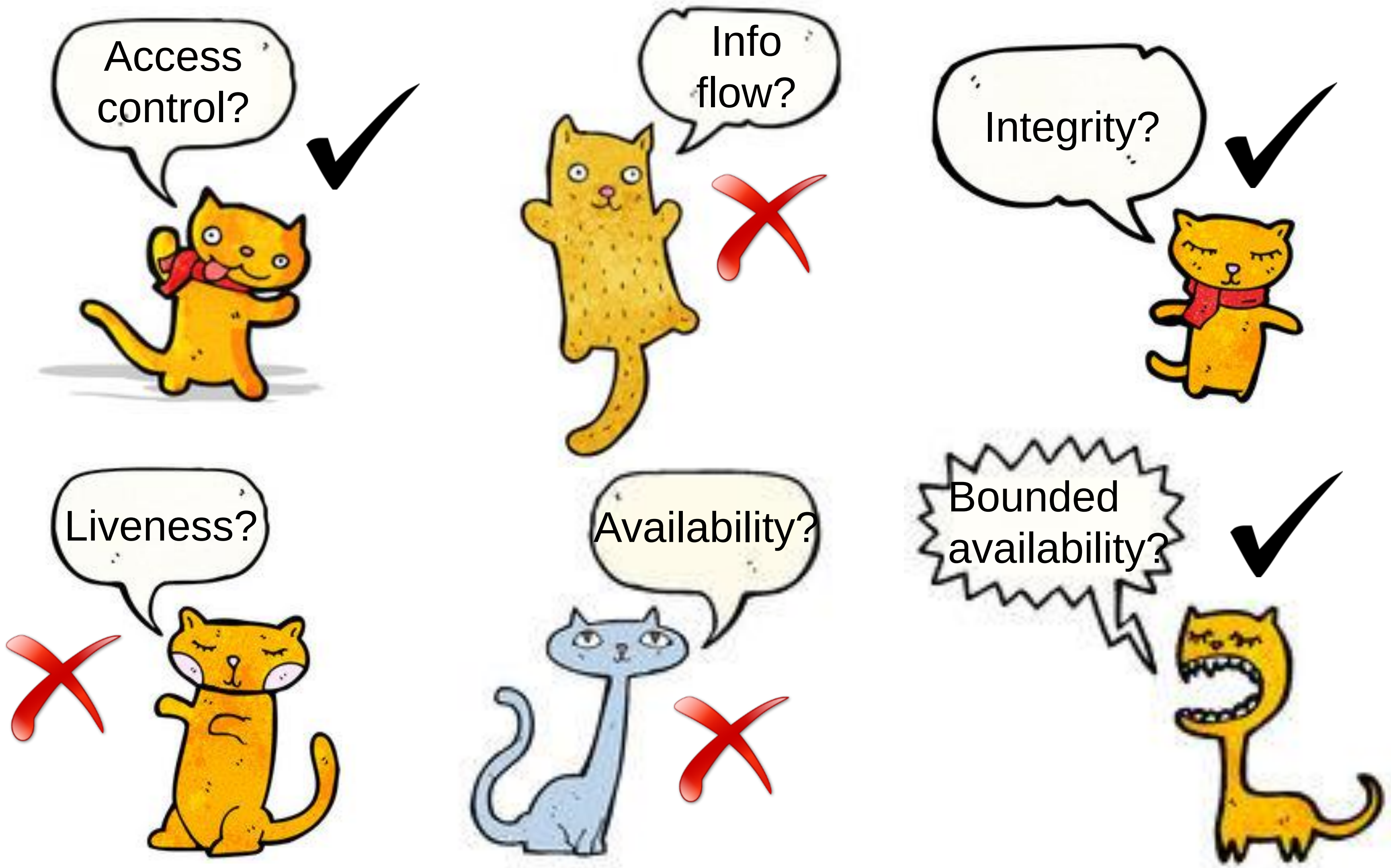
Slide source: Vitaly Shmatikov

- Reference monitors enforce safety policies [Schneider '98]
  - 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"

Some definitions

- **Access control.** “Only my mom can see my Facebook posts.”
- **Information flow.** “Only my mom can see *any value* derived from my Facebook posts.”
- **Integrity.** “Only my mom is allowed to write on my Facebook wall.”
- **Liveness. “**Facebook shows all my Facebook posts to my mom.”
- **Availability.** “The Facebook site is never down for my Mom.”

Which are safety properties?



Some different kinds of safety

Slide source: Vitaly Shmatikov

- Memory safety: all memory accesses are "correct"
  - Respect array bounds, don't stomp on another process's memory, separation between code and data
- 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
    - … but wait until we see mimicry attacks
- Type safety: all function calls and operations have arguments of correct type

Policy enforcement

Slide source: Vitaly Shmatikov

- Checking before every instruction is an overkill
  - Check “No division by zero” only before DIV
- There is a “semantic gap” between individual instructions and policy-level events
  - Applications use abstractions such as strings, types, files, function calls, etc.
  - Reference monitor synthesizes these abstractions

Part Three: Examples of Reference Monitors

CFI: control-flow integrity

[Abadi et al.]

Slide source: Vitaly Shmatikov

- 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

Slide source: Vitaly Shmatikov

- Use binary rewriting to instrument code with runtime checks
- 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

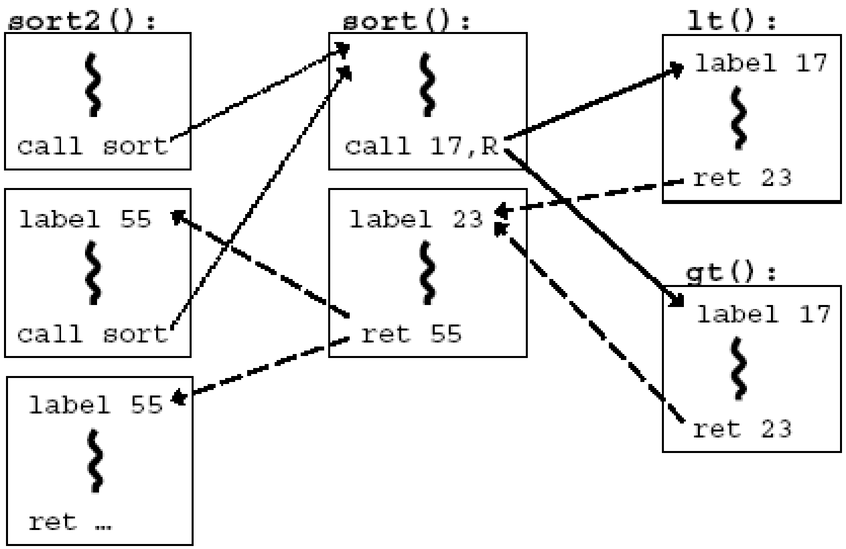
CFG Example

Slide source: Vitaly Shmatikov

```
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

Slide source: Vitaly Shmatikov

- 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

Slide source: Vitaly Shmatikov

Original code

Source Opcode bytes	Instructions	
FF E1	jmp ecx	; computed jump

Destination Opcode bytes	Instructions	
8B 44 24 04	mov eax, [esp+4]	; dst

Instrumented code

```
B8 77 56 34 12     mov   eax, 12345677h     ; load ID-1
40                 inc   eax                ; add 1 for ID
39 41 04           cmp   [ecx+4], eax       ; compare w/dst
75 13              jne   error_label        ; if != fail
FF E1              jmp   ecx                ; jump to label
```

```
3E 0F 18 05     prefetchnta           ; label
78 56 34 12         [12345678h]       ;     ID
8B 44 24 04     mov   eax, [esp+4]    ; dst
...
```

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

Slide source: Vitaly Shmatikov

- 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

CFI: Security Guarantees

Slide source: Vitaly Shmatikov

- 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

WIT: Write Integrity Testing

[Akritidis et al.]

Slide source: Vitaly Shmatikov

- 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 <u>statically valid destinations</u>
    - 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

Slide source: Vitaly Shmatikov

- 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

Slide source: Vitaly Shmatikov

- 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

Slide source: Vitaly Shmatikov

- 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 – read the paper!

WIT: Handling Libraries

Slide source: Vitaly Shmatikov

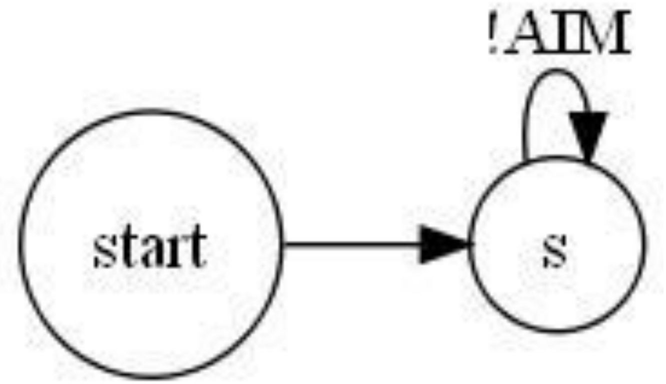
- Basic WIT doesn’t work for libraries (why?)
- Instead, assign the same, standard color to all unsafe objects allocated by library functions and surround them by guards
  - Different from the colors of safe objects and guards
  - Prevents buffer overflows
  - What attack does this not prevent?
- Wrappers 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

Part Four: A Group Exercise

Example from introduction

- **Desired safety property:** Jean never goes on AOL Instant Messenger.
- **Enforcement mechanism:** Operating system and browser enforce the policy.

Security automata for example



Proving correct enforcement

1. Complete mediation.
2. Target control.
3. Isolation.

Discussion: levels of abstraction

Implementation plan?

- Operating system?
- Browser?
- Other mechanisms of information release?

Further reading

Enforceable Security Policies

FRED B. SCHNEIDER
Cornell University

Further reading

Recognizing safety and liveness*

Bowen Alpern[1] and Fred B. Schneider[2]

[1] IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598, USA
[2] Department of Computer Science, Cornell University, Ithaca, NY 14853, USA