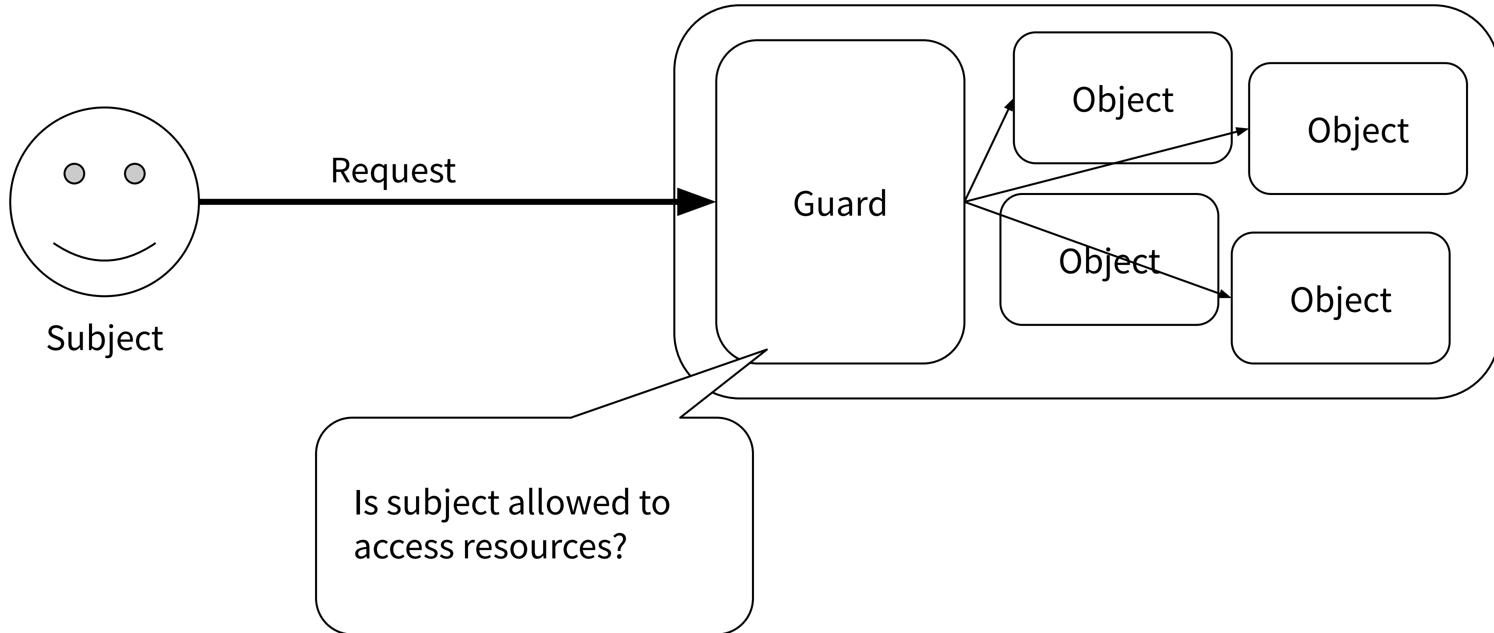
Securing Access to Resources

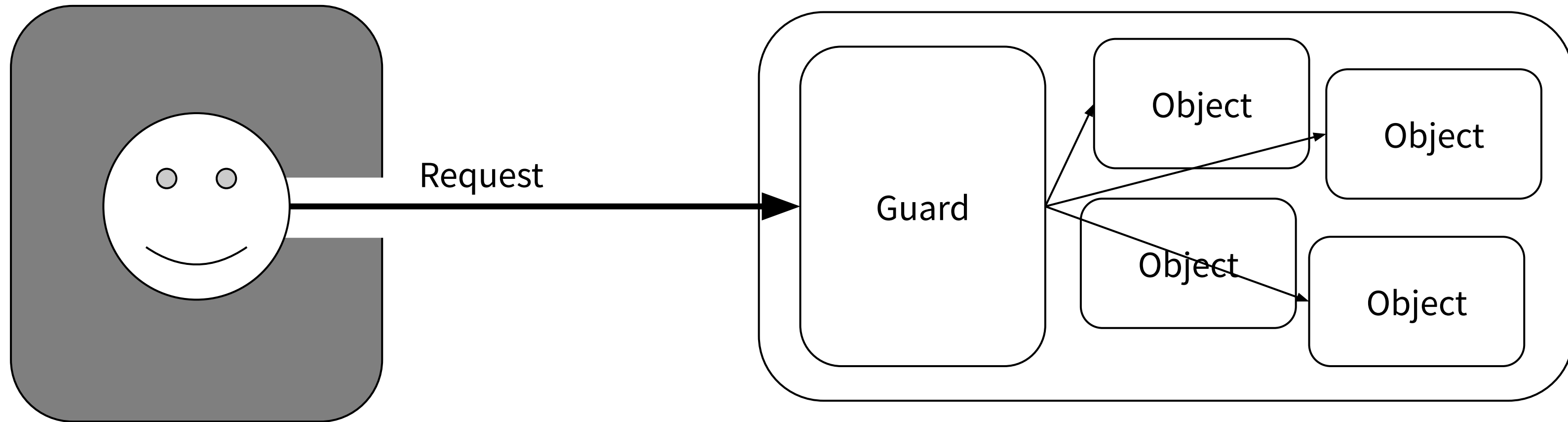
COS 316: Principles of Computer System Design

Amit Levy & Ravi Netravali

Recall The Guard Model



How do we *enforce* the guard model?



Isolation!

Categories of Isolation Mechanisms

- Hardware-Enforced Isolation
  - Memory protection hardware
  - IO/MMU
- Language-based Isolation
  - Dynamic, using interpreters
  - Static, using type-checkers
- Software-Based Fault Isolation
  - Combination of static verification of binary and dynamic checks

Hardware Isolation

How does HW enable isolation?

- Typically, some mechanism to segment memory
    - Virtual memory
        - Sometimes multiple levels of virtualization
    - Segmentation
    - Memory protection hardware
- Operating systems generally use these to create process-like abstractions
    - Each process/service/etc has dedicated memory region for
        - Stack(s)
        - Heap
        - Global variables
    - Process is an isolated address space *and* an execution thread
    - Isolate memory and performance

Hardware Isolation, the good and bad

- Very general purpose
  - Assumes almost nothing about the program (just binary)
- Get performance isolation “for free”
- Normal execution is fast
  - Address translation implemented in hardware, using caches, so typically no overhead
  - E.g., a virtual machine runs compute-heavy workload nearly as fast as a native process
- But, relatively limited
  - Limited hardware resources for isolation
- And communication is expensive
  - Changing protection domains requires heavyweight context switching

Language-isolation

How do languages enable isolation?

- A type-system can restrict which resources a program has access to
- It’s about ***names***
  - If a program cannot ***name*** a resource, there is no way for it to directly access it!
- Key idea: use language features that compartmentalize programs to isolate components
  - E.g. modules, namespaces, closures…
  - Components still share overall memory region (stack, heap, etc), and isolated at finer granularity
- Examples:
  - JavaScript in the browser
  - WebAssembly
  - eBPF
  - SQL queries

Language Isolation, the good and bad

- Virtually infinite isolation granularity
  - Language constructs are often “free”
- Communication can be very cheap
  - Just something like a function call
- Can get strong static guarantees
  - Programs won’t fail at runtime due to access violations
- Limited expressiveness
  - Gotta write your programs in JavaScript!
- Can be slower for normal execution
- Doesn’t isolate performance “for free”

Software Fault Isolation

Software Fault Iso*what*?

- Originally proposed by Wahby et al in 1993
  - Used in practice on-and-off over the years
  - Early versions of VMWare virtual machines, until x86 supported virtualization hardware
  - Native Client in Chrome, until WebAssembly superseded it
- Transforms object code to constraint the data and code it can access
- Best of both worlds?
  - “Arbitrary” binary code
  - Doesn’t need hardware mechanisms, like virtual address spaces (sort of)
- Key idea:
  - Statically analyze the binary
  - Most code, easy to verify it’s fine
  - When an instruction is ambiguous (e.g. an indirect jump, a data-dependent memory access)
    - Wrap instruction in a security monitor

SFI Isolation, the good and bad

- *Pretty* general purpose
  - *Most* binaries can work, but in practice might require special compiler or other restrictions
- Get performance isolation “for free”
  - Similar to HW isolation, all memory can be isolated, including stack
- Normal execution pretty fast, but depends on “unsafe” instructions
- In principle unlimited granularity
  - In practice relocating arbitrary binaries is hard
- Communication not free, but relatively cheap
  - No hardware context switch (don’t need to flush the TLB)
  - Typically requires a jump table and dynamic checks on arguments, so a few extra instructions