Language-based Security

Calculi and Models for Security

Design principles for security

Security Design (I)

***Least Privilege**: each principle is given the minimum access needed to accomplish its task.*

Example:

System Administrators do not run day-to-day tasks as root.

"`rm -rf /`" won't clear the file system!!.

Security Design (II)

Keep the Trusted Computing Base small.

Trusted Computing Base (TCB):

- the parts of a system that must work correctly to ensure the proper functioning of the system.
- e.g., the OS Kernel.

Smaller, simpler systems tend to have fewer bugs and bad interactions.

Principles … today

- The principles of least privilege and small TCB are still valid.
- But today context (wide area network apps) demands for new policies and new enforcement mechanisms.

Security Policies

- Security Policies
  - Protocols (the first part of this series of lectures)
  - Operating systems (undergraduate courses)
  - ….
- Many attacks are high-level, or application-level (such as email worms that pass OS access controls pretending to be executed on behalf of a mailer application).
- Defending against application-level attacks is application-level security
  - language-based security.

What Is Language-based Security?

- Use programming-language and compiler design techniques to enforce software security
- Not just (but also) designing new languages!
- Analyze run-time behaviour
- Analyze existing source codes
- Analyze object codes when sources unavailable

An Example: Memory Safety

- Memory safety: Programs may not access unallocated memory addresses
- Traditional security model:
  - Program is a black box
  - OS kernel intercepts every memory access
- Language-based security model:
  - analyze the program to identify potential violations
  - insert dynamic memory checks into program, if needed

History-based Policies

- No network-sends after file-reads
- Language-based approach: Reference Monitors
  - perform an automated program transformation:
  - inject a new state info: `send_ok:=1`
  - after each file-read instruction, add an assignment: `send_ok:=0`
  - before each send instruction, add a guard: `if (!send_ok) then throw SecurityException`

Information-flow Policies

- Don't divulge my credit card number
- Traditional approach:
  - monitor outgoing network traffic
  - block any transmission containing the relevant bit sequence
- Language-based approach:
  - analyze dataflow of program
  - reject flows from high-security sources to low-security destinations (e.g., network sends)

Semantic analysis

- Examine syntactic properties of code.
  - easy but trivial (e.g. the absence of send ops).
- We want to focus on potential solutions that are based more strongly on the *semantics* or *behavior* of the code.

Some Advantages

- Efficiency
  - avoid unnecessary security checks
  - Certified compilers
- Flexibility
  - no need for custom OS kernel policies
- Rich variety of security policies
  - history-based policies,
  - information flow policies
- Large class of applications
  - Secure orchestration of services
  - Apps for mobile devices

Decidability

- Is language-based security really possible with a static analysis?
- The halting problem
  - reduce memory safety to the halting problem
- Escape (theory-based hackings)
  - limit the domain (e.g. Java only)
  - dynamic checks
  - Sound but incomplete techniques

Language-based Security Research

- Academia:
  - Cornell, Carnegie Mellon, MIT, Princeton, Harvard, Chalmers, INRIA, DIKU… Pisa ….
- Industry:
  - Microsoft, Intel, IBM, ….
  - Mobile phone vendors (e.g., DoCoMo)

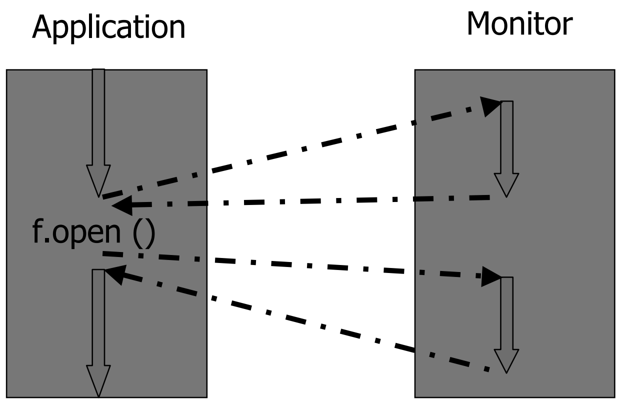
Static vs dynamic

- protect a host from untrusted applications analyzing or modifying application behavior
  - static mechanisms (analysis at compile time)
    - type checking, proof checking, abstract interpretation
  - dynamic mechanisms (analysis at run time)
    - access-control lists, stack inspection, check permissions

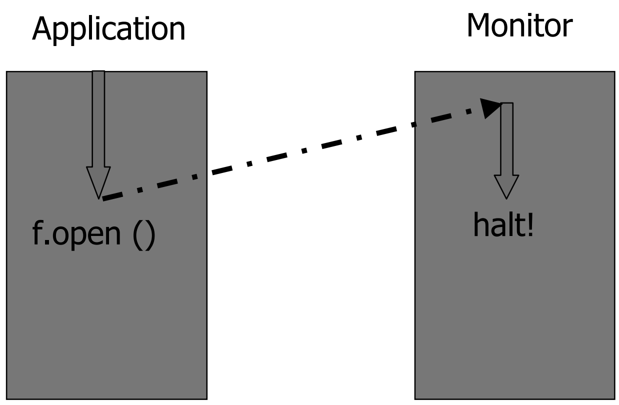
Program Monitors

- A program (or execution or reference) monitor is a module that runs in parallel with an application
  - monitors may detect, prevent, and recover from application errors at run time
  - monitor decisions may be based on execution history

“Good” Operations



“Bad” Operations



Program Monitors: Options

- A program monitor may do any of the following when it recognizes a dangerous operation:
  - halt the application
  - suppress (skip) the operation but allow the application to continue
  - insert (perform) some computation on behalf of the application (e.g. update state information `send_ok:=0`)

Monitors: Security Policies

- Lots of security policies
  - memory safety, control-flow integrity
  - access control policies
  - history-based policies
  - confidentiality (information flow)
- Crucial questions
  - Is there a mathematical framework for defining security policies?
  - What class of security policies is enforceable by some method?
  - Are some enforcement implementations "sound&complete" for some class of policies?

What is a security policy?

The Cornell-Harward School (Schneider, Morrisett, Walker, Harper, ….

Security Policies

- The semantics of a program P is a set of executions [[P]]
- A security policy φ is a predicate on sets of executions.
- A program P satisfies the security policy φ when φ([[P]])
- φ is a trace policy if

$$\varphi(\Sigma) \text{ iff } \forall \sigma \in \Sigma: \varphi(\sigma)$$

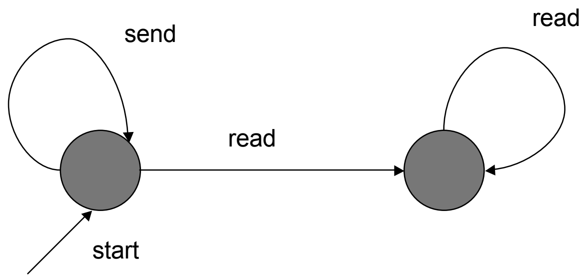
EM Enforceable Policies

1. P satisfies the security policy φ iff φ is satisfied by each individual execution (φ is a trace policy)
2. Execution must be truncated as soon as the sequence violates the policy
   - Violate&recover is not possible
   - Policies are prefix-closed
3. Violations must be detected in a finite amount of time
   - EM can only observe finite executions

EM-Enforceable

- EM enforceable policies are safety properties
- Well-known fact of linear time temporal logic
  - Safety policies can be characterized by (Buchi) automata

Automata and EM-Policies



NO SEND AFTER READ

Computability Results (I)

- A security policy φ of program P is statically enforceable if there exists a TM M that takes an encoding of φ and P as input and, if φ([[P]]) holds, then M accepts in finite time; otherwise M rejects in finite time.
- The class of statically enforceable security properties is the class of recursively decidable properties of programs.

Computability Results (II)

- Safety policies
  - some “bad” thing never happens
  - Example: no out-of-bounds memory access
- Accept program P iff $\forall$ i . (P is good for i steps) where “P is good” must be a decidable property
- This problem is co-r.e. The semi-characteristic function χ_φ(P) that outputs 1 if P is *not* safe on φ, and diverges otherwise:

$$\chi_\varphi(P) \Uparrow \qquad \text{if } \varphi([[P]])$$
$$\chi_\varphi(P) = 1 \qquad \text{otherwise (ie there is a violation)}$$

is computable. The corresponding TM is an execution monitor for φ.

Computability Results (III)

- Liveness Policies:
  - some “good” thing eventually happens
  - Example: lock is eventually released
- Accept program P iff $\exists$ i P does the good thing on step i where “P does the good thing” is decidable
- This problem is recursively enumerable. The semi-characteristic function χ_φ(P) that outputs 1 if P is live on φ, and diverges otherwise:

$$\chi_\varphi(P) = 1 \qquad \text{if } \varphi([[P]])$$
$$\chi_\varphi(P) \Uparrow \qquad \text{otherwise}$$

is computable.

Summary

- For every Safety Policy, there is an Execution Monitor that enforces it.
- There is no known mechanism for *precisely* enforcing a Liveness Policy.
  - “Conservative” (non-precise) enforcement is possible, but only by enforcing some safety policy that approximates the policy of interest
  - Bounded liveness = “good thing will happen within *k* steps” is a safety property

information flow

- My password is not “leaked” to an attacker
- Roughly
  - divide events into “observable events” and “non-observable events”
  - divide input string into “high-security” portion and “low-security” portion
  - P satisfies info flow policy if there is no correlation between high-security input i and observable events exhibited by P(i)
- Info Flow is NOT EM-enforceable
  - No individual execution is “bad” or “good”. Only entire sets of executions are “bad” or “good”.

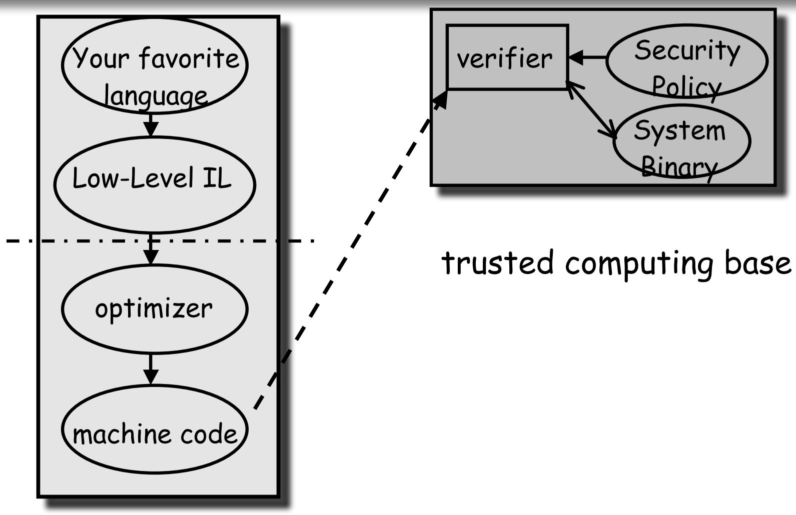
Static vs dynamic

- language-based security mechanisms protect a host from untrusted applications by analyzing or modifying application behavior
  - static mechanisms (analysis at compile time)
    - type checking, proof checking, abstract interpretation
  - dynamic mechanisms (analysis at run time)
    - access-control lists, stack inspection, check permissions

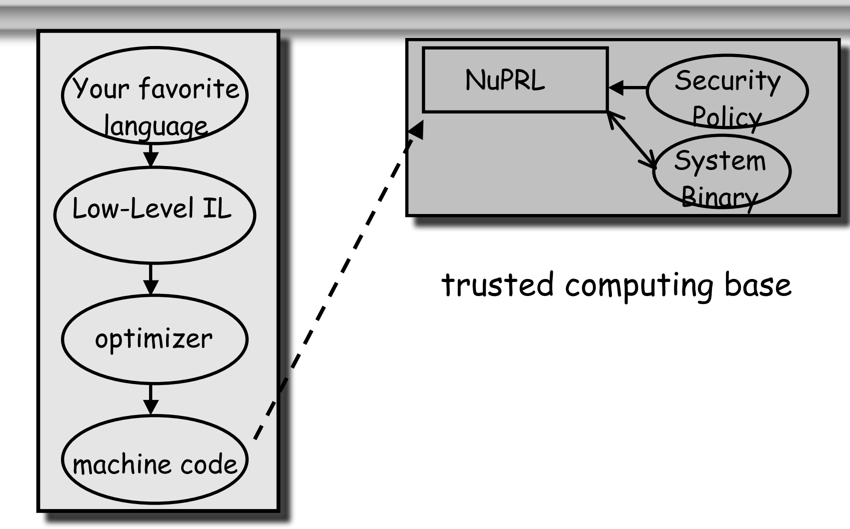
Proof Carrying Code

Necula, Lee, Appel, etc

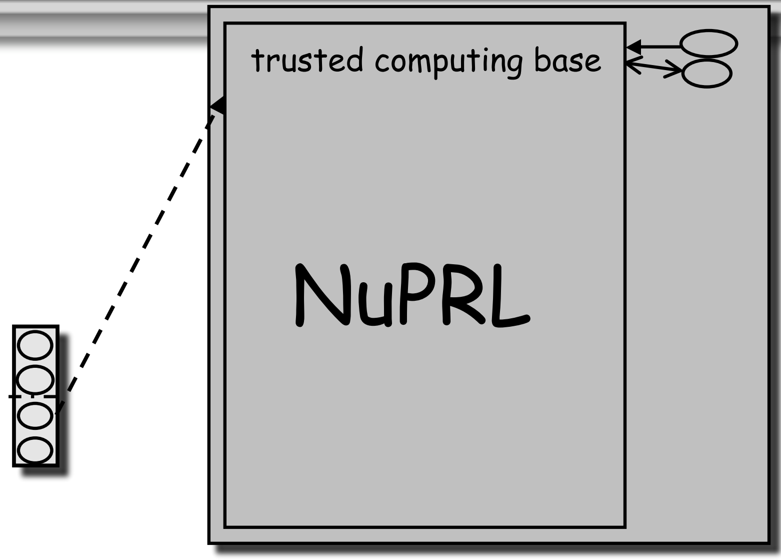
The ideal world



Theorem Prover

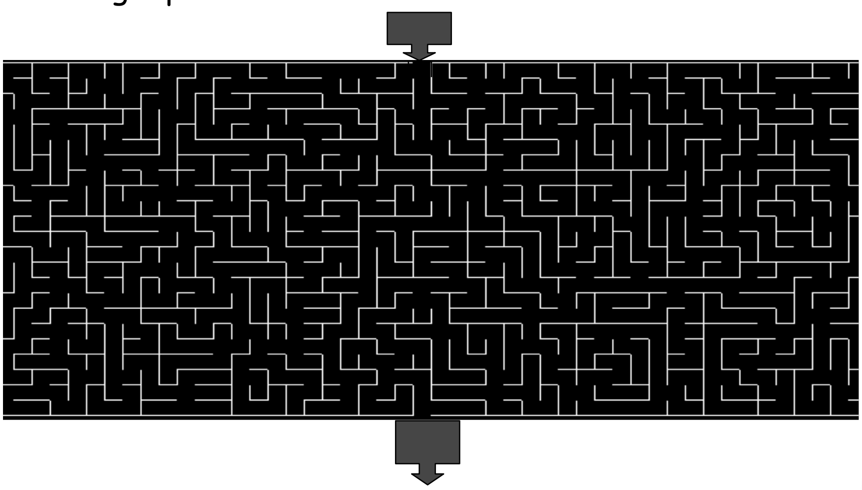


Unfortunately…



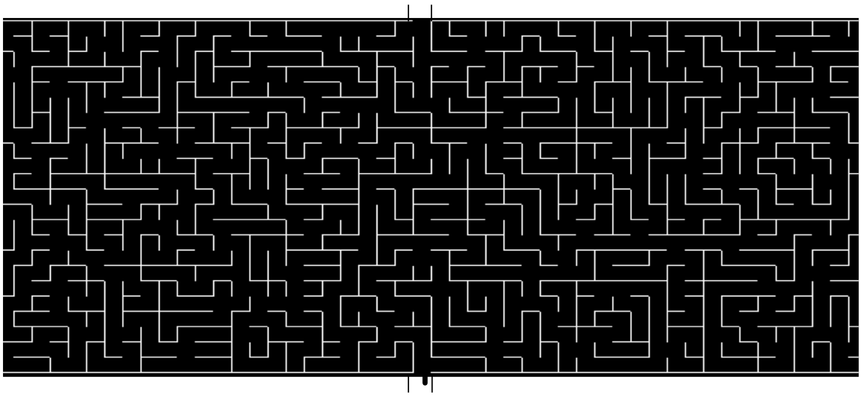
PCC Observation

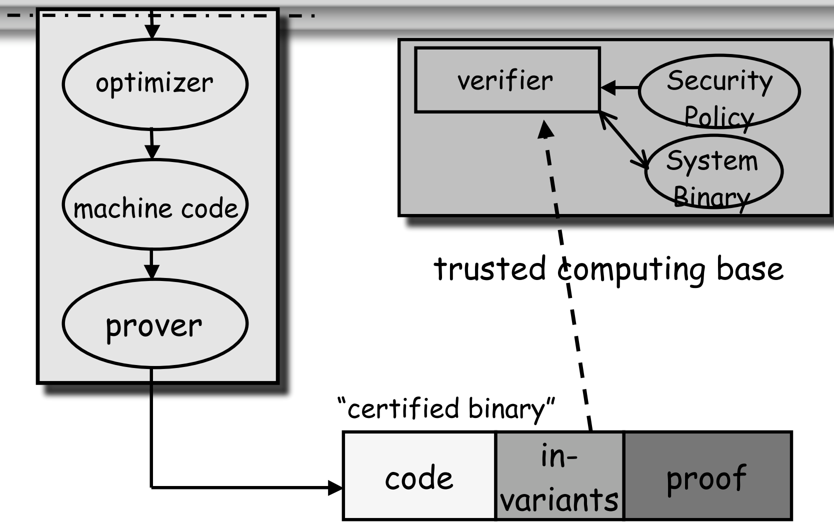
Finding a proof is hard …..



PCC Observation

Finding a proof is hard, but verifying a proof is easy.





Verification

The Verifier (~ 4-6 pages of C code):

- takes code, loop invariants, and policy
- calculates the verification condition A.
- checks that the proof is a valid proof of A:
  - fails if some step doesn't follow from an axiom or inference rule
  - fails if the proof is valid, but not a proof of A

Advantages of PCC

In Principle:

- Simple, small, and fast TCB.
- No external authentication or cryptography.
- No additional run-time checks.
- Precise and expressive specification of code safety policies.

Summary

- Proof-carrying code is a great principle.
  - For special-purpose applications (eg java cards) is extremely efficient.
- General-purpose extensions:
  - Need some way to get the proof automatically (limit policy to type-safety).
  - Engineering proof size is an issue.
  - Compiling high-level languages is an issue.

Information Flow

Type System

Volpano, Myers, Sabelfeld, Sands, ….

Information Flow

- Information flow policy asserts that secret input data cannot be inferred by an attacker via attacker's observations
- Non interference: a program is secure iff high inputs do not interfere with low level view of the system

Static Approach

- Track information flow via suitable types
  - Program correctness via static type checking
- (Simplified) Assumption
  - High variable (secret)
  - Low variable (public)
  - Low level obs does not reveal high level data

Examples

l := h	Not secure (direct)
l:= h ; l:= 0	Secure
h := l ; l:= h	Secure
if h = 0 then l:=0 else l:= 1	Not secure (indirect flow)
While h=0 do skip	Secure

Program Semantics

- [[P]]: Int x Int --> Int x Int

Semantics

$[[P]] : Int \times Int \dashrightarrow (Int \times Int)_{\perp}$

Non termination

High Input

Low Input

High Output

Low Output

Semantics: Examples

[[l := h]](s,s’)	(s,s’)
[[l:= h ; l:= 0]] (s,s’)	(s,0)
[[if h = 0 then l:=0 else l:= 1]] (1,s’)	(1,1)
[[while h=0 do skip]] (0,s’)	⊥

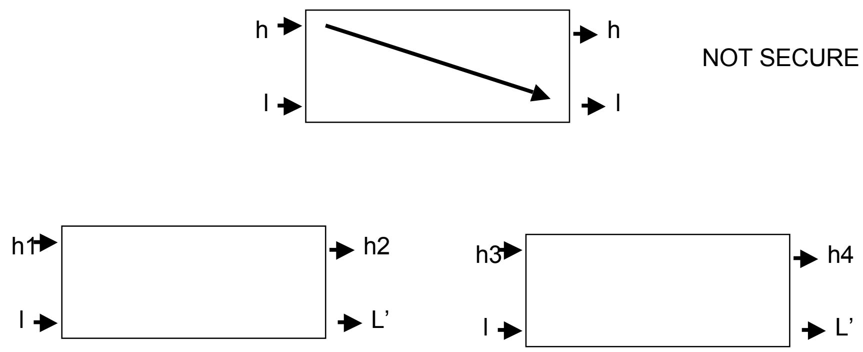
Semantic Analysis

Low Level Equality
$(h, l) =_L (h', l')$ iff $l=l'$

Low Level View
$S \approx_L S'$ iff $\perp \neq S$ implies $S =_L S'$

P is ***SECURE***
iff
$\forall$ S, S', $S =_L S'$ implies $[[\ P\ ]]S \approx_L [[\ P\ ]]S'$

Graphically



SECURE

Secure type system

- Expressions
  - exp: high
  - h $\notin$ Var(exp) $\Rightarrow$ exp: low
- Atomic Statements
  - [pc] |- skip
  - [pc] |- h:= exp
  - exp: low $\Rightarrow$ [low] |- l:= exp

Secure Type System

- [high] |- C $\Rightarrow$ [low] |- C
- [pc] |- C, [pc] |- C' $\Rightarrow$ [pc] |- C ; C'
- e: [pc], [pc] |- C, [pc] |- C' $\Rightarrow$
  [pc] if e then C else C'
- e: [pc], [pc] |- C, $\Rightarrow$ [pc] while e do C

Typing: Examples

[low] |- h:=l+4 ; l:= l-5;
[pc] |- if h then h := h+7 else skip
[low] |- while l < 34 do l:= l+1
[pc] |/- while h<4 do l:=l+1

Exercize

- [?] |- h:=h+1 ;
  if l=0 then l:=5 else l:3

Exercize

- [low] |- h:=h+1 ;
  if l=0 then l:=5 else l:3

Soundness

- Soundness Theorem

  [pc] |- C ⇒ C is secure

More …

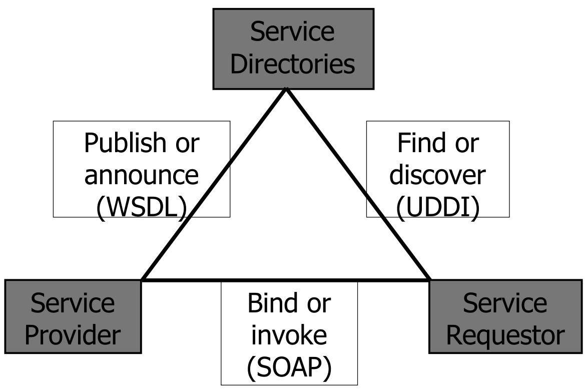
- Procedure
- Concurrency
- Non determinism
- Distribution
- Exception
- Declassification
- ….

Challenge

- Putting all these things together
  - Local policies
  - Type System
  - Code Certification

Service Oriented Computing

The SOC Triangle



Service Interfaces

The description should be unambiguous, formal representations of

- A service's functionality
- A service's nonfunctional attributes

Discovery & Composition

1. Finding the right services
   1. Semantic matchmaking
2. Service composition is the core sw issue in SOA.
   1. Applications are created by combining the basic building blocks provided by other services.
   2. Service compositions may themselves become services, following a model of recursive service composition..
3. Many composition models are possible.
   1. Process oriented composition – Orchestration
   2. Distributed composition – Choreography
4. Security issues
   1. TCB
   2. Ws security (low lovel properties)

- Massimo & Roberto next week!!!