CIS 551 / TCOM 401

Computer and Network Security

Spring 2007
Lecture 24

Announcements

- Project 4 is Due Friday April 20th at 11:59 PM
- Final exam:
  - Friday, May 4th. 9:00 - 11:00 a.m. Towne 313
- Thursday's Class:
  - Review
  - Project 4
  - Course evaluations (please come!)

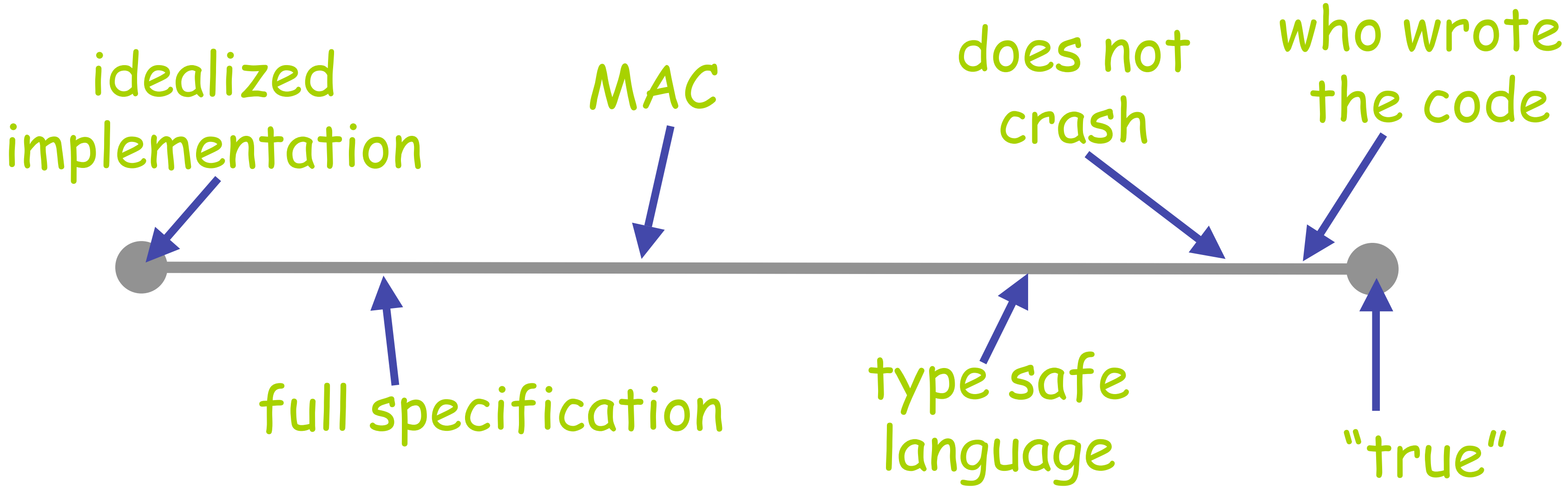
What is “Bad”?

Depends upon:

- Task: what is the program’s purpose?
- Context: what host, OS, whose behalf?
- Policy: e.g., mandatory access control

Tighter constraints are better? Sometimes.

No silver bullet.



Trends:

Vendors	few	→	many
Media	hard	→	soft
Delivery mechanism	physical	→	electronic
Frequency of installation	seldom	→	always
Size of package	whole thing	→	small pieces
Permanence	persistent	→	ephemeral

Challenges

- Complexity of the software
  - # components going up
  - everything is extensible
  - legacy C and C++ code to interact with
- Complexity of policy
  - Internet has complicated trust models
    - many more parties involved
    - much more dynamic systems
  - More confidential information online
  - More exposure to attack
- ⇒ Need for tools to *improve* security of software, both for producers & consumers

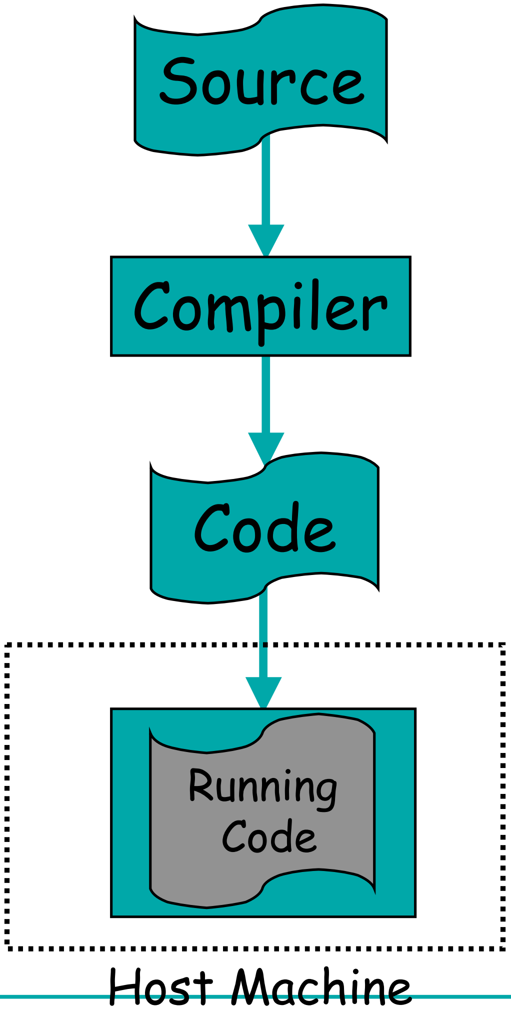
Language-based Tools for Security

- Birds-eye view of some new technologies
  - Protect software consumers (end-users) from malicious programs
  - Help software developers create more robust, secure programs

- Measuring security?

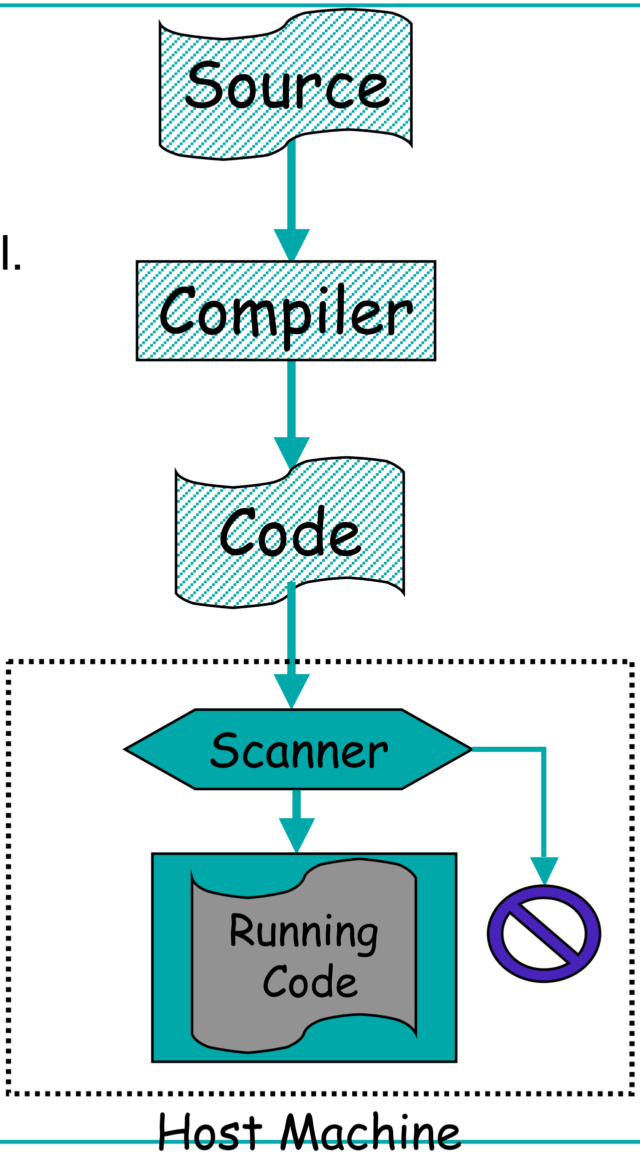
Software Deployment Architecture

- Trusted Computing Base
  - Becomes huge when software is run on many, many hosts
- Minimumize TCB:
  - Ensure the quality of the software
- Must be cheap, easy to deploty
  - Otherwise won't be adopted



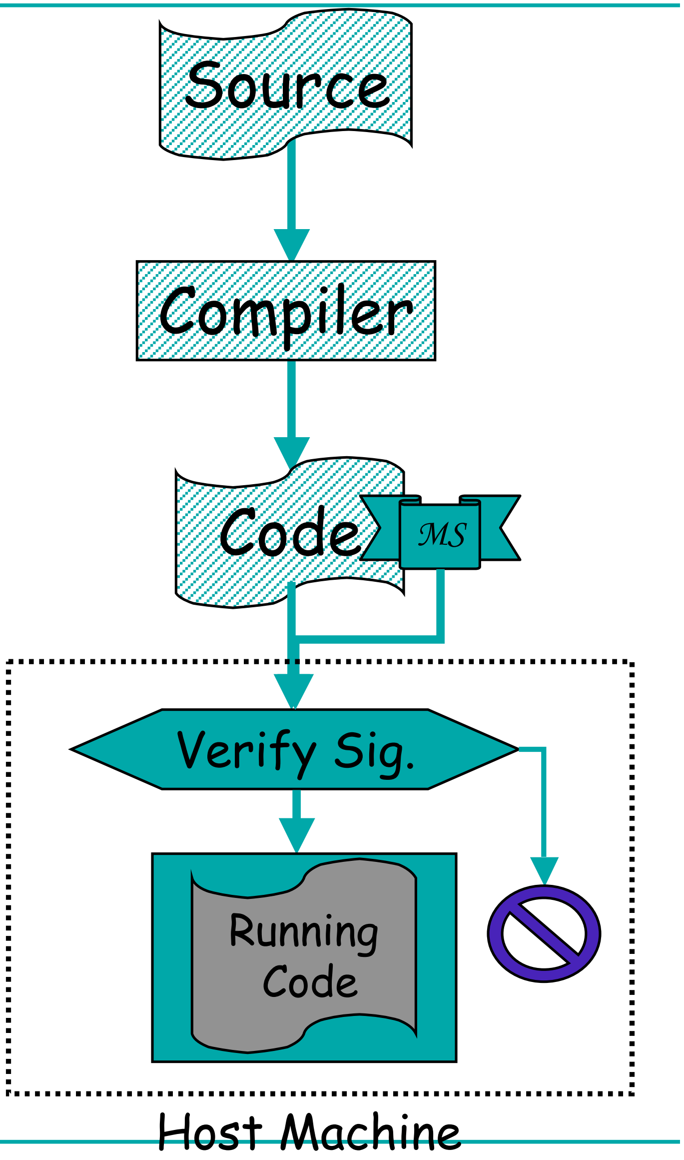
Existing Approach: Virus Scanners

- Virus Scanners?
  - e.g., McAffee, Norton, etc.
  - perhaps the most commercially effective tool.
  - only works for previously seen bad code.
  - virus kits make it easy to disguise a virus.
  - not clear that it scales over time.
- Not a complete solution



Existing Approach: Signatures

- Digital Signatures of Code?
  - e.g., Verisign, Authenticode, MS device drivers
  - bad assumption: signature implies "good"
    - keys may be stolen
    - "good" for what context?
    - even well-intentioned people make "bad" code
  - bad assumption: you can sue the signer
- Not a complete solution
- Can we do better?



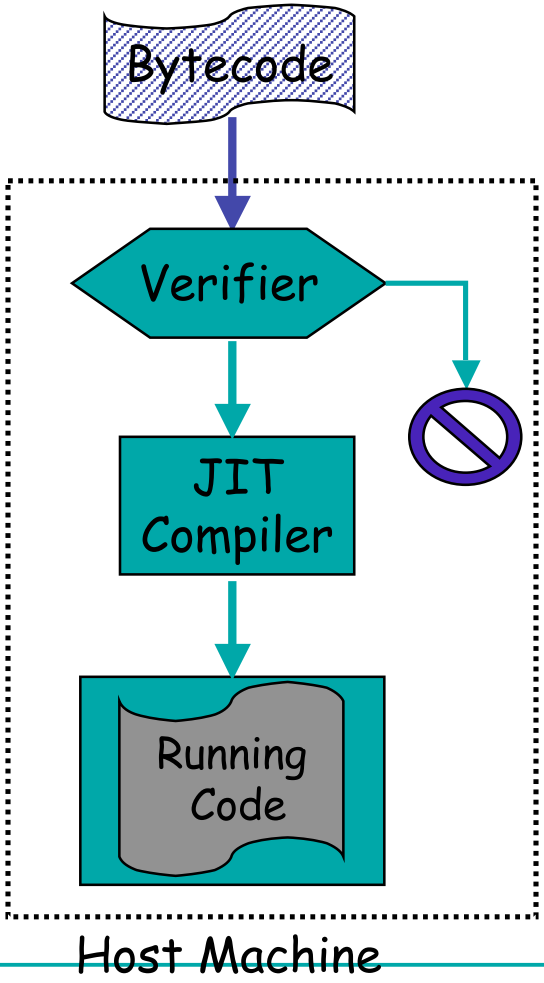
Language-based Security

- Use compiler & programming language technology to improve security.
- Before the program runs
  - Proof Carrying Code (PCC)
  - Jif - Java for Information Flow

- During the program execution
  - Inlined Reference Monitors

Java Bytecode

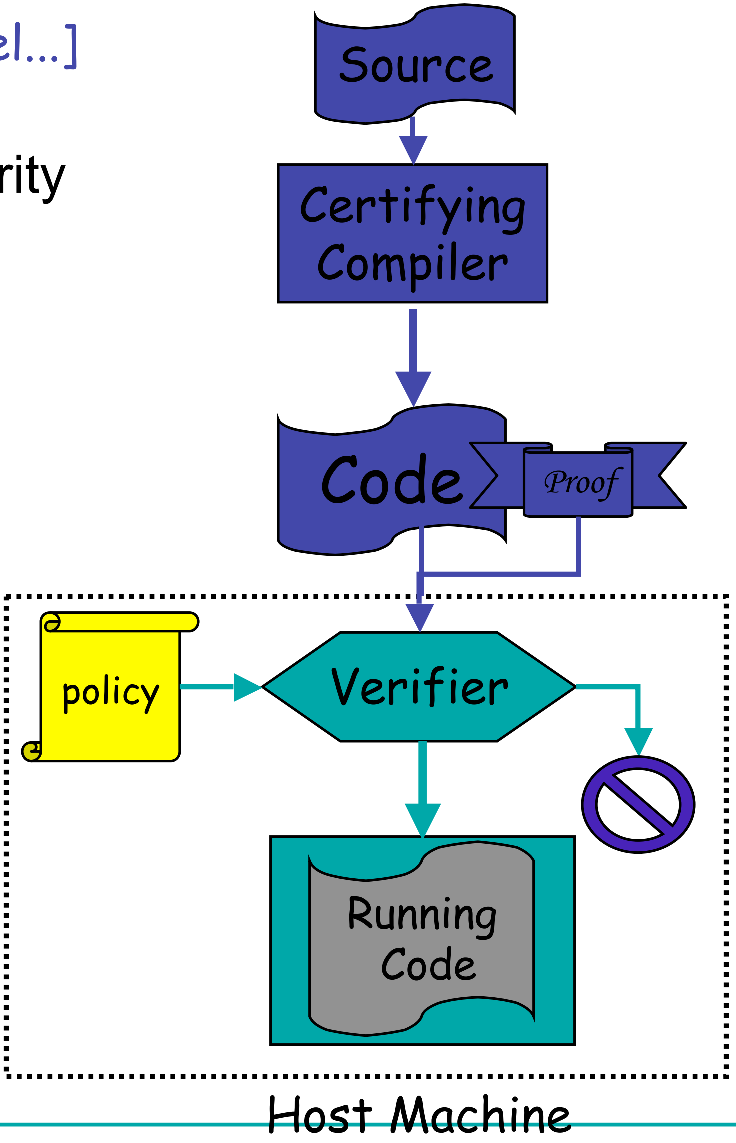
- Verify the bytecode at the consumer
- Pro: Simple, cost effective
- Con: Large TCB:
  - commercial, optimizing JIT: 200,000-500,000 LOC
  - when is the last time your favorite software company wrote a bug-free 200,000 line program?
- Con: Java specific policy



Proof Carrying Code

[Necula & Lee '97, Morrisett '98, Appel...]

- Verify a *provided* proof of program security
  - Meaning of the proof connected to meaning of program (unlike signatures)
  - Up to code producer to generate proof
  - Consumer only has to *check* the proof
- Verifier is *small*
  - 3000 LOC



Source

Certifying Compiler

Code

Proof

policy

Verifier

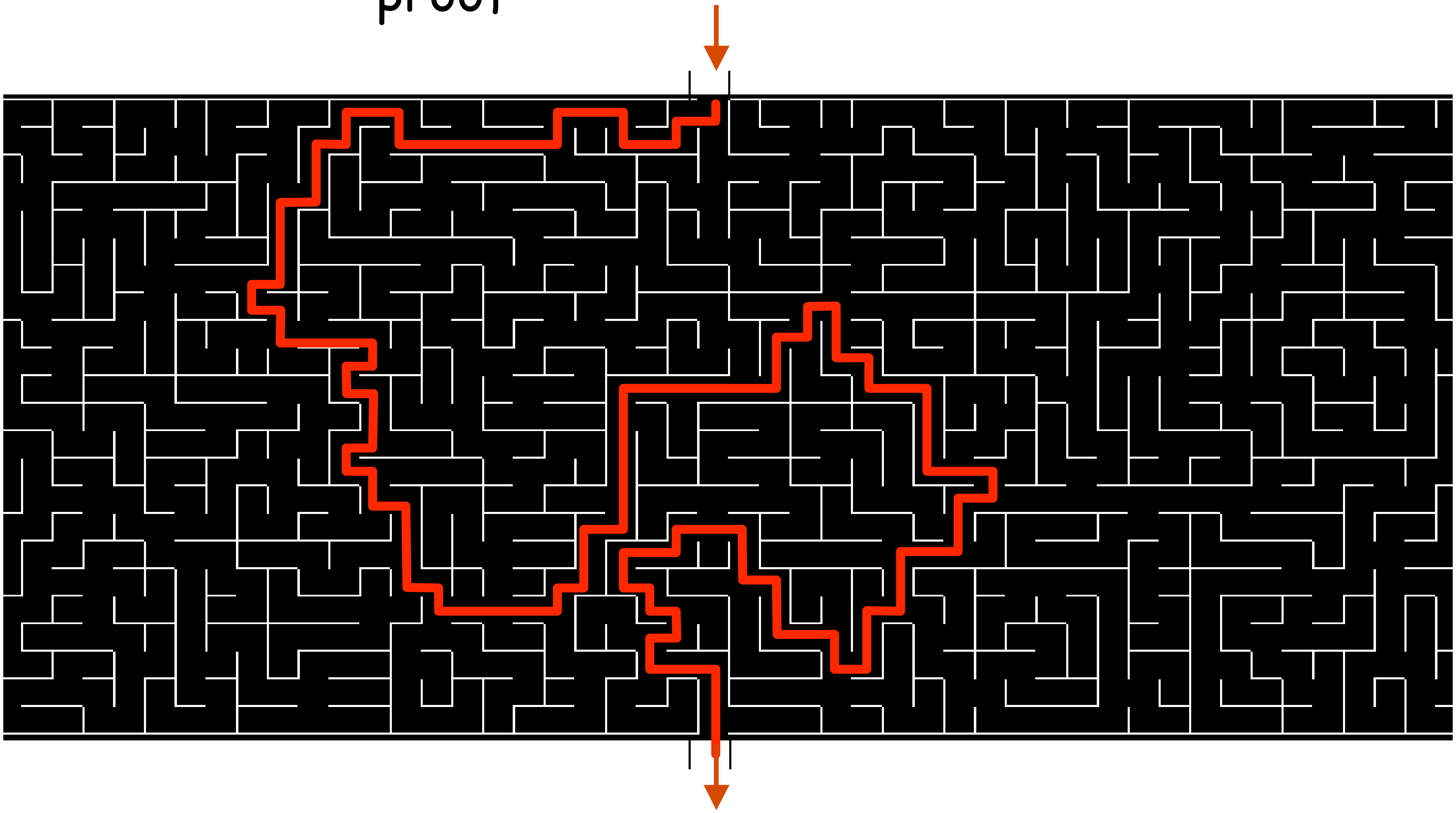
Running Code

Host Machine

PCC: An Analogy

Legend: ┐┘ code

— proof



PCC Advantages

- Reduces the TCB
  - Verification is simpler/faster than proof generation.
  - Consumer is independent of how the proof is generated ⇒ compiler not trusted.
- Tamperproof
  - Changing the proof or program is either (1) detected or (2) proven to be OK.
- No cryptography, no trusted 3rd party
- No run-time overhead
  - Static checking

PCC Engineering Challenges

- Where do you get the proof?
  - Programmer & compiler
  - Automated techniques needed
- Dealing with formal proofs
  - Must be machine checkable
  - Naive encoding of proofs of program properties are very large.
    - Careful engineering reduces overhead
- Touchstone Compiler [Necula & Lee]
  - Java to Intel x86 assembly language
  - Enforces Java's security policy without byte code interpreter or large trusted JIT

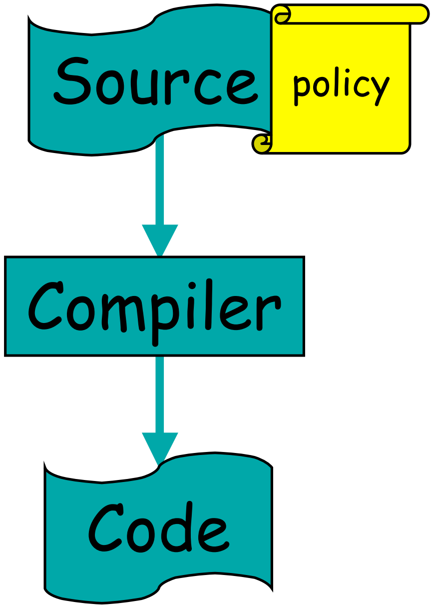
Security-oriented Languages

- PCC doesn't address policy
  - type safety ⇒ no crashes
  - in principle, can enforce any policy
  - ... but how to describe the policy?
- Programming languages with facilities for implementing specific policies
  - Confidentiality
    - protect secrets
  - Integrity
    - prevent tampering
  - Availability
    - ensure legitimate use succeeds

Jif = Java + Information Flow

[Myers, Zdancewic, Zheng, Chong, Nystrom]

- Problem: Lots of confidential info.
  - passwords, e-mail, financial data, medical data, buisiness transactions, ...
- Existing technology essential, but...
  - OS doesn't provide fine grained control
  - Cryptography not the solution
  - Not "end-to-end" solutions
- Philosophy: improve security, do not try to eliminate covert channels
  - Modern take on MLS security



Security Policies in Jif

- Confidentiality labels:
  - `int{Alice:} x;` "Alice's private int"
  - `int{Alice:Bob}y;` "Alice permits Bob as reader"
- Integrity labels:
  - `int{*:Alice} z;` "Alice trusts z"
- Combined labels:
  - `int{Alice: ; *:Alice} w;` (Both)

```
int{Alice:} a1, a2;
int{Bob:} b;
int{*:Alice} c;
```

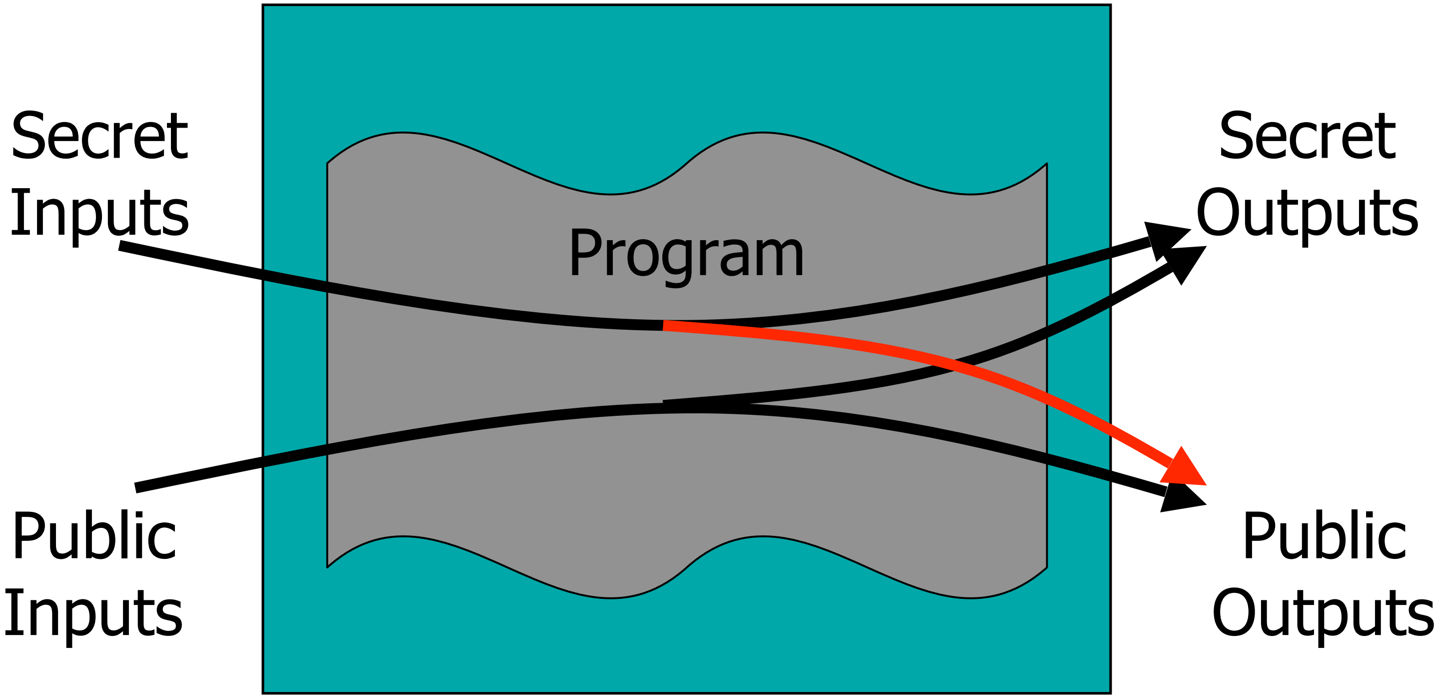
Insecure

```
a1 = b;
b = a1;
c = a1;
```

Secure

```
a1 = a2;
a1 = c;
```

Information Confidentiality



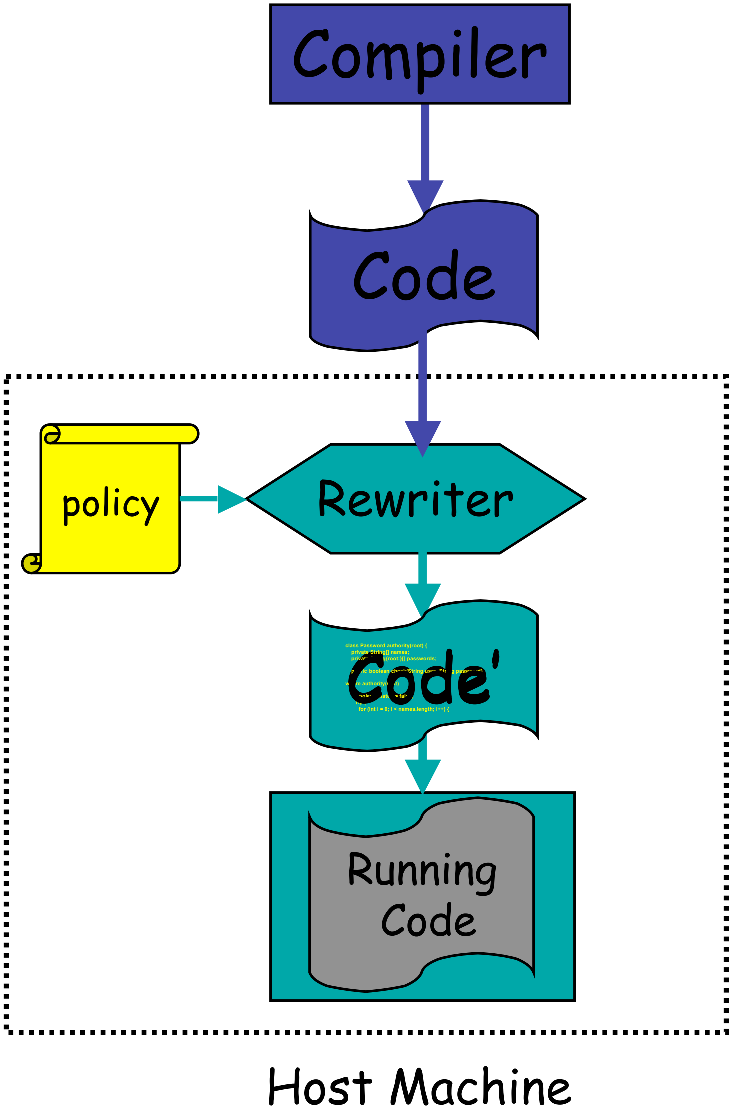
Jif Advantages

- Explicit information-flow policies
  - compiler checks program for compliance
- Finer granularity than OS
- Enforces rich, programmable policies
  - e.g “Medical data should not be sent to the public printer.”
  - e.g. “Financial data should be encrypted before being transmitted over the Internet.”
- Permits end-to-end security
- Similar technology already or soon to be used:
  - Perl: Prevents “bad” data from being used inappropriately (lightweight MLS)
  - Microsoft e-mail will control dissemination

Inlined Reference Monitors

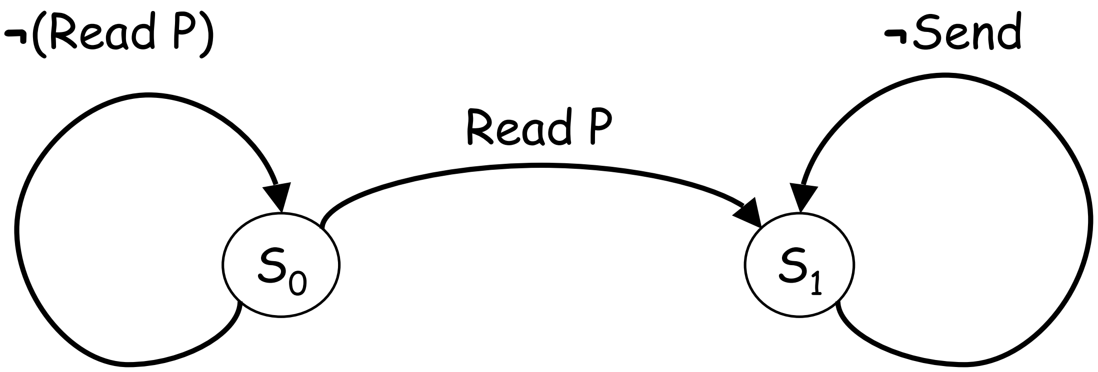
[Schneider & Erlingsson]

- Rewrite the code at the consumer's machine
  - Have the system administrator specify a policy.
  - Transform the untrusted code to obey the policy

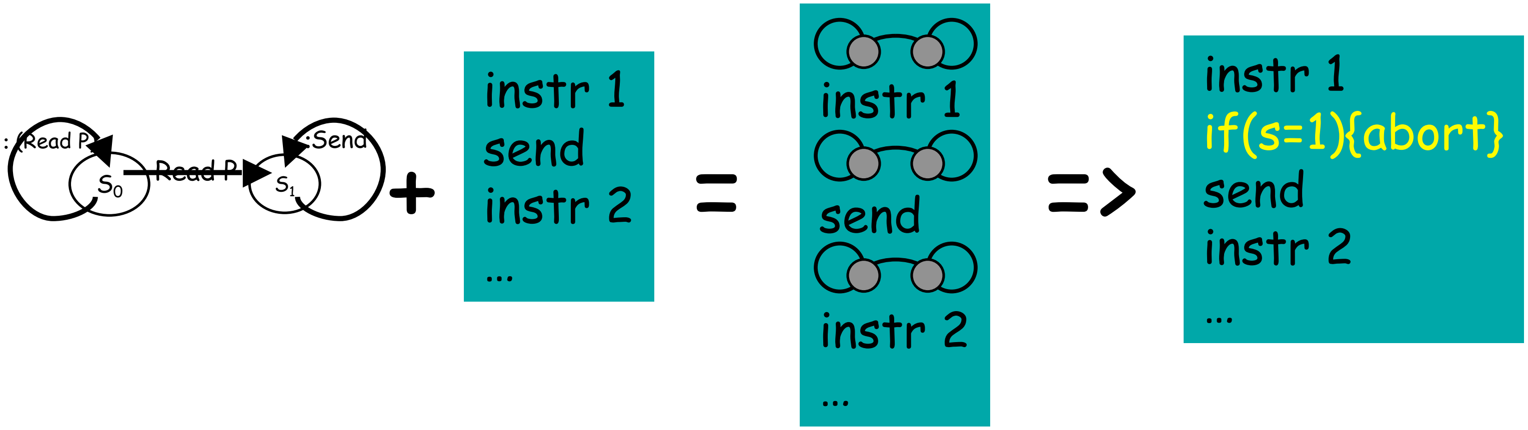


IRM: Example Policy

"No network sends after private file P has been read."



IRM: Code instrumentation



```
instr 1
send
instr 2
…
```

```
instr 1
send
instr 2
…
```

```
instr 1
if(s=1){abort}
send
instr 2
…
```

- Conceptually:
  - Evaluate the reference monitor in parallel with the program
- Implemented by adding state
- Checking state before each instruction
  - Optimize to eliminate overhead

IRM Advantages

- Consumer does not have to trust the software
- Can be made very efficient
- Once policy is determined, deployment can be automatic
- Flexible
  - Implemented Java stack inspection

- Disadvantage:
  - Sometimes difficult to describe high-level policies in terms of low-level operations like assembly language instructions

PL Technology Summary

- Proof Carrying Code
  - Robust & scalable security infrastructure
  - Flexible policy mechanisms
- Security-oriented languages (Jif)
  - End-to-end confidentiality & integrity
  - Explicit policies mean understanding tradeoffs
- Inlined reference monitors
  - Efficiently monitor the behavior of applications
- Java / C# just the start!

Authorization Logics

- An authorization logic is a domain-specific language for writing access-control policies [ABLP]
- Logical connectives:

T ::= true | c | α | $T \wedge T$ | $T \vee T$ | $T \rightarrow T$ | $\forall\alpha.T$ |
P says T

- Define "P speaks-for Q" = $\forall\alpha$. (P says α) → (Q says α)
  - (Q says (P speaks-for Q)) → (P speaks-for Q)
    "Q can delegate its authority to P" (The "hand off" axiom)

- Example proposition:

(f:File, FS says may-read(Q,f))

"f is a file and the FS says that principal Q may read f"

Authorization Logic Programming Model

- Processes as reference monitors:
  - Make access control decisions based on policies expressed in this authorization logic.
- Processes as clients:
  - Create and pass evidence (in the form of proofs) that they are authorized to perform certain actions.
  - Analogous to the "capabilities" discussed in the access control part of the course
- Information-flow control:
  - Control the flow of information through the reference monitor.
- Decentralized / distributed implementation:
  - Possible proof that "P says T" is P's digital signature on a string "T"
  - Associate a private key with each process (the "authority" of the process)

An example program

```
getOwner : (f:File) → ∃O.FS says owns(O,f)

send : ∀O,R. (f:File) →
             O says mayRead(R,f) →
             FS says owns(O,f) → true

readReq = ∃A,R. R says {f:File; A says mayRead(R,f)}

handleRead(readReq r){
    let  {A;R;req} = r;
    bind {f;c} = req in
      let {O;ownP} = getOwner(f);
      check ownP:(FS says owns(A,f)) {        // note: O=A
        send [A,R] f c
      }
}
```

What about cost & performance?

- Tragedy of the commons
  - Everyone would benefit from better security
  - Market forces are disincentive to build secure software
    - Time to ship often outweighs security (and even correctness)
    - “The user’s going to choose dancing pigs over security every time.” – Bruce Schnier
- Java/C# are slower than C, but…
  - Type safety ⇒ no crashes
  - Array bounds checks ⇒ no buffer overflows
  - Garbage collection ⇒ no memory management errors
- Security-oriented languages are promising, but…
  - Still in the research stages
  - How usable in practice?