PLDI'06 Tutorial T1: Enforcing and Expressing Security with Programming Languages

Andrew Myers
Cornell University
http://www.cs.cornell.edu/andru

Computer security

- Goal: prevent bad things from happening
  - Clients not paying for services
  - Critical service unavailable
  - Confidential information leaked
  - Important information damaged
  - System used to violate laws (e.g., copyright)
- Conventional security mechanisms aren't up to the challenge

Harder & more important

In the '70s, computing systems were isolated.

- software updates done infrequently by an experienced administrator.
- you trusted the (few) programs you ran.
- physical access was required.
- crashes and outages didn't cost billions.

The Internet has changed all of this.

- we depend upon the infrastructure for everyday services
- you have no idea what programs do.
- software is constantly updated – sometimes without your knowledge or consent.
- a hacker in the Philippines is as close as your neighbor.
- everything is executable (e.g., web pages, email).

Language-based security

- Conventional security: program is black box
  - Encryption
  - Firewalls
  - System calls/privileged mode
  - Process-level privilege and permissions-based access control
- Prevents addressing important security issues:
  - Downloaded and mobile code
  - Buffer overruns and other safety problems
  - Extensible systems
  - Application-level security policies
  - System-level security validation
- Languages and compilers to the rescue!

Outline

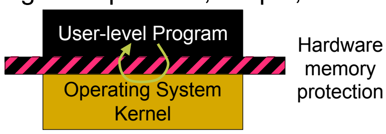
- The need for language-based security
- Security principles
- Security properties
- Memory and type safety
- Encapsulation and access control
- Certifying compilation and verification
- Security types and information flow

- Handouts: copy of slides
- Web site: updated slides, bibliography
  www.cs.cornell.edu/andru/pldi06-tutorial

Security principles

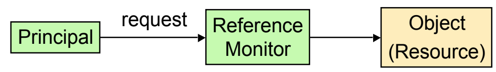
Conventional OS security

- Model: program is black box
- Program talks to OS via protected interface (system calls)
  - Multiplex hardware
  - Isolate processes from each other
  - Restrict access to persistent data (files)
- \+ Language-independent, simple, limited



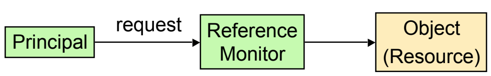
Access control model

- The classic way to prevent "bad things" from happening
- Requests to access resources (objects) are made by principals
- Reference monitor (e.g., kernel) permits or denies request



Authentication vs. Authorization

- Abstraction of a principal divides enforcement into two parts
  - Authentication: **who** is making the request
  - Authorization: is this principal allowed to make this request?



1st guideline for security

Principle of complete mediation:

Every access to every object must be checked by the reference monitor

Problem: OS-level security does not support complete mediation

OS: Coarse-grained control

- Operating system enforces security at system call layer
  - Hard to control application when it is not making system calls
- Security enforcement decisions made with regard to large-granularity objects
  - Files, sockets, processes
- Coarse notion of principal:
  - If you run an untrusted program, should the authorizing principal be "you"?

Need: fine-grained control

- Modern programs make security decisions with respect to *application* abstractions
  - UI: access control at window level
  - mobile code: no network send after file read
  - E-commerce: no goods until payment
  - intellectual property rights management
- Need extensible, reusable mechanism for enforcing security policies
  - Language-based security can support an extensible protected interface, e.g., Java security

2nd guideline for secure design

Principle of Least Privilege: *each principal is given the minimum access needed to accomplish its task.* [Saltzer & Schroeder '75]

Examples:

- + Administrators don't run day-to-day tasks as root. So "`rm -rf /`" won't wipe the disk.
- - fingerd runs as root so it can access different users' .plan files. But then it can also "`rm -rf /`".

Least privilege problems

- OS privilege is coarse-grained: user/group
- Applications need finer granularity
  - Web applications: principals unrelated to OS principals
- Who is the "real" principal?
  - Trusted program? Full power of the user principal
  - Untrusted? Something less
  - Trusted program with untrusted extension: ?
  - Untrusted program accessing secure trusted subsystem: ?
- Requests may filter through a chain of programs or hosts
  - Loss of information is typical
  - E.g., client browser → web server → web app → database

3rd guideline: Small TCB

Trusted Computing Base (TCB) : components whose failure compromises the security of a system

- Example: TCB of operating system includes kernel, memory protection system, disk image
- Small/simple TCB:
  - ⇒ TCB correctness can be checked/tested/reasoned about more easily ⇒ more likely to work
- Large/complex TCB:
  - ⇒ TCB contains bugs enabling security violations
  - Problem: modern OS is huge, impossible to verify

Small TCB and LBS

- Conventional wisdom (c. 1975):
  - "operating system is small and simple, compiler is large and complex"
  - OS is a small TCB, compiler a large one
- c. 2003:
  - OS (Win2k) = 50M lines code, compiler ~ 100K lines code
  - Hard to show OS implemented correctly
    - Many authors (untrustworthy: device drivers)
    - Implementation bugs often create security holes
  - Can now prove compilation, type checking correct
    - Easier than OS: smaller, functional, not concurrent

The Gold Standard [Lampson]

- **Au**thenticate
  - Every access/request associated with correct principal
- **Au**thorize
  - Complete mediation of accesses
- **Au**dit
  - Recorded authorization decisions enable after-the-fact enforcement, identification of problems
- Language-based techniques can help

When to enforce security

Possible times to respond to security violations:

- Before execution:
  - analyze, reject, rewrite
- During execution:
  - monitor, log, halt, change
- After execution:
  - roll back, restore, audit, sue, call police

Language-based techniques

A complementary tool in the arsenal: programs don't have to be black boxes! Options:

1. Analyze programs at compile time or load time to ensure that they are secure
2. Check analyses at load time to reduce TCB
3. Transform programs at compile/load/run time so that they can't violate security, or to log actions for auditing.

Maturity of language tools

- How to build a sound, expressive type system that provably enforces run-time type safety
  ⇒ protected interfaces
- Type systems that are expressive enough to encode multiple high-level languages
  ⇒ language independence
- How to build fast garbage collectors
  ⇒ trustworthy pointers
- On-the-fly code generation and optimization
  ⇒ high performance

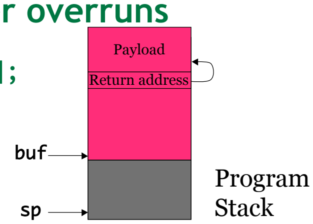
Caveat: assumptions and abstraction

- Arguments for security always rest on assumptions:
  - "the attacker does not have physical access to the hardware"
  - "the code of the program cannot be modified during execution"
  - "No one is monitoring the EM output of the computer"
- Assumptions are vulnerabilities
  - Sometimes known, sometimes not
- Assumptions arise from *abstraction*
  - security analysis only tractable on a simplification (abstraction) of actual system
  - Abstraction hides details (assumption: unimportant)
- Caveat: language-based methods often abstract aspects of computer systems
  - Need other runtime, hardware enforcement mechanisms to ensure language abstraction isn't violated—a separation of concerns

A sampler of attacks

Attack: buffer overruns

```
char buf[100];
…
gets(buf);
```



- Attacker gives long input that overwrites function return address, local variables
- "Return" from function transfers control to payload code

Execute-only bit?

- Stack smashing executes code on stack -- mark stack non-executable?
- Return-to-libc attack defeats this:

```
void system(char * arg) {
      ...
      r0 = arg;
      execl(r0, ...); // "return" here with r0 set
      ...
}
```

  - Not all dangerous code lives in the code segment…
- More attacks: pointer subterfuge (function- and data-pointer clobbering), heap smashing, overwriting security-critical variables…
- Moral: SEGVs can be turned into attacks

Attack: format strings

```
fgets(sock, s, n);
…
fprintf(output, s);
```

- Attack: pass string **s** containing a **%n** qualifier (writes length of formatted input to arbitrary location)
- Use to overwrite return address to "return" to malicious payload code in **s**.

Attack: SQL injection

- Web applications typically construct SQL database queries.
  - In PHP:

```
$rows=mysql query("UPDATE users SET pass='$pass'
  WHERE userid='$userid'");
```

  - Attacker uses userid of `' OR '1' = '1'`. Effect:

```
UPDATE users SET pass=<pass> WHERE userid='' OR '1'='1'
```

- 69% of Internet security vulnerabilities are in web applications [Symantec]

Using system subversion

- Assume attacker can run arbitrary code (possibly with dangerous privileges)
- Initial foothold on target system enables additional attacks (using other holes)
- Worms: programs that autonomously attack computers and inject their own code into the computer
- Distributed denial of service: many infected computers saturate target network

1988: Morris Worm

Penetrated an estimated 5 to 10 percent of the 6,000 machines on the internet.

Used a number of clever methods to gain access to a host.

- brute force password guessing
- bug in default sendmail configuration
- X windows vulnerabilities, rlogin, etc.
- buffer overrun in fingerd

Remarks:

- System diversity helped to limit the spread.
- "root kits" for cracking modern systems are easily available and largely use the same techniques.

1999: Love Bug & Melissa

Both email-based viruses that exploited:

- a common mail client (MS Outlook)
- trusting (i.e., uneducated) users
- VB scripting extensions within messages to:
  - lookup addresses in the contacts database
  - send a copy of the message to those contacts

Melissa: hit an estimated 1.2 million machines.

Love Bug: caused estimated $10B in damage.

Remarks:

- no passwords or crypto involved

Why did it succeed?

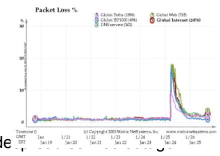
- Visual Basic scripts invoked transparently upon opening
- Run with full privileges of the user
- Kernel doesn't know about fine-grained application abstractions or related security issues: mail messages, contacts database, etc.
- Recipients trusted the sender – after all, they know them
- Interactions of a complex system were unanticipated

A solution for Melissa?

- Turn off all executable content?
  - no problem when email was just text.
  - but executable content is genuinely useful.
  - ex: automated meeting planner agent, postscript, Mpeg4 codecs, client-side forms, etc.
  - US DoD tried to do this : revolt
- Fundamental tension:
  - modern software wants to be open and extensible.
  - *programmable* components are ultimately flexible.
    - Postscript, Emacs, Java[script], VB, Jini, ActiveX, plug-n-play...
  - security wants things to be closed: least privilege.
  - turning off extensibility is a denial-of-service attack.

2002: MS-SQL Slammer worm

- Jan. 25, 2002: SQL and MSDE servers on Internet turned into worm broadcasters
  - Buffer-overrun vulnerability
  - Spread to most vulnerable servers on the Internet in less than 10 min!
- Denial of Service attack
  - Affected databases unavailable
  - Full-bandwidth network load ⇒ wide
  - "Worst attack ever" – brought down many sites, not Internet
- Can't rely on patching!
  - Infected SQL servers at Microsoft itself
  - Owners of most MSDE systems didn't know they were running it…extensibility again



Virus scanning?

- Scan for suspicious code
  - e.g., McAfee, Norton, etc.
  - based largely on a lexical signature.
  - the most effective commercial tool
  - but only works for things you've seen
    - Melissa spread in a matter of hours
  - virus kits make it easy to disguise a virus
    - "polymorphic" viruses
- Doesn't help with worms
  - Unless you can generate a filter automatically…

Security Properties

Security properties

Security = "bad things don't happen"

What kinds of properties should computing systems satisfy?

Security policies

- Execution (trace) of a program is a sequence of states $s_1 s_2 s_3 \ldots$ encountered during execution
  - Program has a set of possible executions T
- A generic formalization: security policy is a predicate P on sets of executions
  - Program satisfies policy if P(T)
- Examples:
  - P(T) if no null pointer is deferenced in any trace in T
  - P(T) if every pair of traces in T with the same initial value for x have the same final value for y

Safety properties

- "Nothing bad ever happens"
- A property is a policy that can be enforced using individual traces
  - P(T) ⇔ ∀t∈T. P'(t) where P' is some predicate on traces
- Safety property can be enforced using only history of program
  - If P'(t) does not hold, then all extensions of t are also bad
  - Amenable to run-time enforcement: don't need to know future
- Examples:
  - access control (e.g. checking file permissions on file open)
  - memory safety (process does not read/write outside its own memory space)
  - type safety (data accessed in accordance with type)

Liveness properties

- "Something good eventually happens"
  - If P'(t) does not hold, every finite sequence t can be extended to satisfy P'
- Example: nontermination
  - "The email server will not stop running"
- Violated by denial of service attacks
- Can't enforce purely at run time
- Interesting properties often involve both safety and liveness
  - Every property is the intersection of a safety property and a liveness property [Alpern & Schneider]

Memory safety and isolation

- *Process isolation*: running process cannot access memory that does not belong to it
  - Usually enforced by hardware TLB
    - TLB caches virtual→physical address mappings
    - Invalid virtual addresses (other processes) cause kernel trap
  - Cross-domain procedure calls/interprocess communication (RPC/IPC) expensive (*TLB misses*)
- *Memory safety:* running process does not attempt to dereference addresses that are not valid allocated pointers
  - No read from or write to dangling pointers
  - Not provided by C, C++ :

```
int *x = (int *)0x14953300;
*x = 0x0badfeed;
```


Control-flow integrity

- Actual control flow must conform to a "legal execution"
- Code injection attacks violate CFI.
- Weak: control can only be transferred to legal program code points
  - Rules out classic buffer overrun attacks
  - Not provided by C:

```
int (*x)() = (int(*)()) 0xdeadbeef; (*x)();
```

- Stronger: control must agree with a DFA or CFG capturing all legal executions
- Can be enforced cheaply by dynamic binary rewriting as in DynamoRIO [Kiriansky et al., 2002]

Type safety

- Values manipulated by program are used in accordance with their types
  - Stronger than memory safety!
- Can be enforced at run-time (Scheme), compile-time (ML), mix (Java)
- Abstract data types: data types that can only be accessed through a limited interface
  - can protect their internal storage (private data)
- Kernel = ADT with interface = system calls, abstraction barrier enforced at run time by hardware

Access control

- Access control decision:
  - principal × request × object → boolean
- Access control matrix [Lampson]:

objects / principals

principals \ objects	file1	file2	file3
user1	r	rw	rx
user2	r	r	
user3	rw	r	

Allowed requests

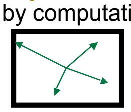
  - Columns of matrix: access control lists (ACLs)
- Correct enforcement is a safety property
  - Safety can be generalized to take into account denial of access, corrective action by reference monitor [Hamlen][Ligatti][Viswanathan]

Information security

- Sometimes computer security is an aspect of physical security
  - Make sure attackers cannot take over electric power distribution grid, military command-and-control, etc.
  - Can use type safety, access control to enforce rules
- What we're trying to protect can also be the information on the computer: information security
  - Memory safety, type safety don't directly help

Information security: confidentiality

- Confidentiality: valuable information should not be leaked by computation.

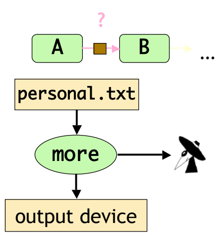


- Also known as *secrecy,* though sometimes a distinction is made:
  - Secrecy: information itself is not leaked
  - Confidentiality: nothing can be learned about information
- Simple (access control) version:
  - Only authorized processes can read from a file
  - But… when should a process be "authorized" ?

Confidentiality: a Trojan horse

- Access control controls release of data but does not control propagation
- Security violation even with "safe" operations

```
% ls -l personal.txt
rw-------   personal.txt
% more personal.txt
...
```

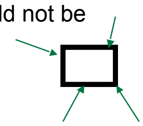


End-to-end confidentiality

- Access control does not help after access control check is done
- End-to-end confidentiality: Information should not be improperly released by a computation no matter how it is used
- Enforcement requires tracking information flow
  - Encryption provides end-to-end secrecy—but prevents most computation

Information security: integrity

- Integrity: valuable information should not be damaged by computation



- Simple (access control) version:
  - Only authorized processes can write to a file
  - But… when should a process be "authorized"
- End-to-end version:
  - Information should not be updated on the basis of less trustworthy information
  - Requires tracking information flow in system
- Information flow is not a property [McLean94]
  - No information flow from x to y:
    $\Leftrightarrow$
    P(T) if every pair of traces in T with the same initial value for x always have the same value for y

Privacy and Anonymity

- Anonymity:
  - individuals (principals) and their actions cannot be linked by an observer
  - alt: identity of participating principals cannot be determined even if actions are known
- Privacy: encompasses aspects of confidentiality, secrecy, anonymity

Availability

- System is responsive to requests
- DoS attacks: attempts to destroy availability (perhaps by cutting off network access)
- Fault tolerance: system can recover from *faults* (failures), remain available, reliable
- Benign faults: not directed by an adversary
  - Usual province of fault tolerance work
- Malicious or Byzantine faults: adversary can choose time and nature of fault
  - Byzantine faults are attempted security violations
  - usually limited by not knowing some secret keys

Enforcing safety properties

Reference Monitor

Observes the execution of a program and halts the program if it's going to violate the security policy.

Common Examples:

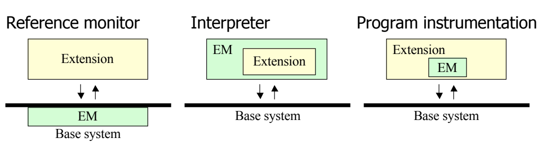
- memory protection
- access control checks
- routers
- firewalls

Most current enforcement mechanisms are reference monitors

Requirements for a Monitor

- Must have (reliable) access to information about security-relevant actions of the program
  - e.g., what instruction is it about to execute?
- Must have the ability to "stop" the program
  - can't stop a program running on a different machine
  - … or transition to a "good" state.
- Must protect the monitor's state and code from tampering.
  - key reason why a kernel's data structures and code aren't accessible by user code
- low overhead in practice

Pervasive mediation



OS Reference monitor: won't capture all events
Wrapper/interpreter: performance overhead
Instrumentation: merge monitor into program

- different security policies ⇒ different merged-in code
- simulation does not affect program
- pay only for what you use

What policies?

- Reference monitors can only see the past
  - They can enforce safety properties but not liveness properties

Assumptions:

- monitor can have access to entire state of computation.
- monitor can have arbitrarily large state
- safety properties enforced are modulo computational power of monitor
- But: monitor can't guess the future – the predicate it uses to determine whether to halt a program must be computable.

Software Fault Isolation (SFI)

- Wahbe et al. (SOSP'93)
- Goal is process isolation: keep software components in same hardware-based address space, provide
  - Idea: application can use untrusted code without memory protection overhead
- Software-based reference monitor isolates components into *logical address spaces*.
  - conceptually: check each read, write, & jump to make sure it's within the component's logical address space.
  - hope: communication as cheap as procedure call.
  - worry: overheads of checking will swamp the benefits of communication.
- Only provides memory isolation, doesn't deal with other security properties: confidentiality, availability,…

One way to SFI: Interpreter

```
void interp(int pc, reg[], mem[], code[], memsz, codesz) {
  while (true) {
     if (pc >= codesz) exit(1);
     int inst = code[pc], rd = RD(inst), rs1 =
  RS1(inst),
        rs2 = RS2(inst), immed = IMMED(inst);
   switch (opcode(inst)) {
        case ADD: reg[rd] = reg[rs1] + reg[rs2]; break;
        case LD:  int addr = reg[rs1] + immed;
                  if (addr >= memsz) exit(1);
                  reg[rd] = mem[addr];
                  break;
        case JMP: pc = reg[rd]; continue;
   ...
  }
   pc++;
}}
```

```
0: add r1,r2,r3
1: ld r4,r3(12)
2: jmp r4
```


Interpreter pros and cons

Pros:

- easy to implement (small TCB.)
- works with binaries (high-level language-independent.)
- easy to enforce other aspects of OS policy

Cons:

- terrible execution overhead (25x? 70x?)

It's a start.

Partial Evaluation (PE)

A technique for speeding up interpreters.

- we know what the code is.
- specialize the interpreter to the code.
  - unroll the main interpreter loop – one copy for each instruction
  - specialize the switch to the instruction: pick out that case
  - compile the resulting code
- Can do at run time with dynamic binary rewriting (e.g., DynamoRIO)
  - Keep code cache of specialized code
  - Reduce load time, code footprint

Example PE

Original Binary:

```
0: add r1,r2,r3
1: ld r4,r3(12)
2: jmp r4
 ...
```

Interpreter

```
case LD:  int addr = reg[rs1] + immed;
          if (addr >= memsz) exit(1);
          reg[rd] = mem[addr];
          break;
```

Specialized interpreter:

```
reg[1] = reg[2] + reg[3];
addr = reg[3] + 12;
if (addr >= memsz) exit(1);
reg[4] = mem[addr];
pc = reg[4]
```

Resulting Code

```
0: add r1,r2,r3
1: addi r5,r3,12
2: subi r6,r5,memsz
3: jab _exit
4: ld r4,r5(0)
…
```


Sandboxing

- SFI code rewriting is "sandboxing"
- Requires code and data for a security domain are in one contiguous segment
  - upper bits are all the same and form a *segment id*.
  - separate code space to ensure code is not modified.
- Inserts code to ensure load and stores are in the logical address space
  - force the upper bits in the address to be the segment id
  - no branch penalty – just mask the address
  - re-allocate registers and adjust PC-relative offsets in code.
  - simple analysis used to eliminate some masks

Jumps

- Inserts code to ensure jump is to a valid target
  - must be in the code segment for the domain
  - must be the beginning of the translation of a source instruction (tricky for variable-length instructions)
- PC-relative jumps are easy:
  - just adjust to the new instruction's offset.
- Computed jumps are not:
  - must ensure code doesn't jump *into* or *around* a check or else that it's *safe* for code to do the jump.

More SFI Details

Protection vs. Sandboxing:
- Protection is fail-stop:
  - stronger security guarantees (e.g., reads)
  - required 5 dedicated registers, 4 instruction sequence
  - 20% overhead on 1993 RISC machines
- Sandboxing covers only stores
  - requires only 2 registers, 2 instruction sequence
  - 5% overhead

Remote (cross-domain) Procedure Call:
- 10x cost of a procedure call
- 10x faster than a really good OS RPC

Sequoia DB benchmarks: 2-7% overhead for SFI compared to 18-40% overhead for OS.

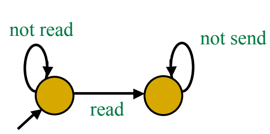
Limitations of SFI

- Only enforces process isolation
- Variable-length instructions are tricky
  - But provably correct SFI is possible for x86 [McCamant & Morrisett]
- Sometimes want to enforce more complex rules on untrusted code
  - Example: downloaded applet can either read local files or send to network, but not both
- Can we do more by code rewriting?

Inlined reference monitors (IRMs)

- SASI [Schneider & Erlingsson 1999], Naccio [Evans & Twyman 1999]
- SFI inlines a particular safety policy into untrusted code
- Idea: embed an arbitrary safety policy into untrusted code at load time
  - Policy may be application-specific, even user-specific
  - Low execution overhead

Security automata

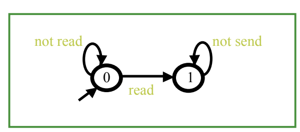


- Every safety property enforceable by security automaton [Schneider '98]
- System execution produces sequence of events … … automaton reads and accepts/rejects sequence
- Need *pervasive mediation* to allow policies independent of code being checked

Example: JVM code in SASI

(Also implemented for x86 machine code)

```
…
ldc 1                                     // new automaton state on stack
putstatic SASI/stateClass/state           // cause automaton state change
invokevirtual java/io/FileInputStream/read()I // read integer from file
…
getstatic SASI/stateClass/state           // automaton state onto stack
ifeq SUCCEED                              // if stacktop=0 goto succeed
   invokestatic SASI/stateClass/FAIL()V   // else security violation
SUCCEED:
   invokevirtual java/net/SocketOutputStream/write(I)V     // send message
...
```



PSLang: *specifying* policies

- State diagrams (SASI) are inconvenient -- how to specify a reference monitor?
- Policy Specification Language (PSLang)
  - same expressive power, more convenient
  - event-driven programming model maps program actions (events) to automaton state updates
  - specification expressible in terms of application abstractions
- Has been used to specify, enforce Java stack inspection security model (!) with good performance
- But..hard to apply complex policies to low-level code

```
STATE { boolean did_read = false; }
EVENT methodCall FileInputStream.read { did_read = true; }
EVENT methodCall Network.send CONDITION did_read { FAIL; }
```


Type Safety and Security

Type-safe languages

Software-engineering benefits of type safety:

- memory safety
- no buffer overruns (array subscript `a[i]` only defined when `i` is in range for the array `a`.)
- no worries about self-modifying code, wild jumps, etc.

- Type safety can be used to construct a protected interface (e.g., system call interface) that applies access rules to requests

Java

- Java is a type-safe language in which type safety is security-critical
- Memory safety: programs cannot fabricate pointers to memory
- Type safety: must use objects at correct types
- Encapsulation: private fields, methods of objects cannot be accessed from outside
- Bytecode verifier ensures compiled bytecode is type-safe

Java: objects as capabilities

- Single Java VM may contain processes with different levels of privilege (e.g. different applets)
- Some objects are *capabilities* to perform security-relevant operations:

```
FileReader f = new
FileReader("/etc/passwd");
// now use "f" to read password file
// …but don't lose track of it!
```


Problems with capabilities

- Original 1.0 security model: use type safety, encapsulation to prevent untrusted applets from accessing capabilities in same VM
- Problem: tricky to prevent capabilities from leaking (downcasts, reflection, …)
  - One approach: confined types [Vitek&Bokowski]
- Difficult to revoke capabilities esp. in distributed environment

Java Stack Inspection

- Added to Java to deal with capability model shortcomings
- Dynamic authorization mechanism
  - close (in spirit) to Unix effective UID
  - attenuation and amplification of privilege
- Richer notion of context
  - operation can be good in one context and bad in another
  - E.g: local file access
    - may want to block applets from doing this
    - but what about accessing a font to display something?

Security operations

- Each method has an associated *protection domain*
  - e.g., applet or local
- `doPrivileged(P){S}`:
  - fails if method's domain does not have priv. P.
  - switches from the caller's domain to the method's while executing statement S (think setuid).
- `checkPermission(P)` walks up stack S doing:

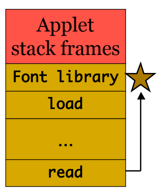
```
for (f := pop(S); !empty(S) ; f := pop(S)) {
  if domain(f) does not have priv. P then error;
  if f is a doPrivileged frame then break;
}
```

- Very operational description! But ensures integrity of control flow leading to a security-critical operation

Example

```
Font Library:
  ...
  doPrivileged(ReadFiles) {
     load("Courier");
  }
  ...
```

```
FileIO:
  ...
  checkPermission(ReadFiles);
  read();
  ...
```



Requires:

- Privilege enabled by some caller (applet can't do this!)
- All code between enabling and operation is trustworthy

Some pros and cons?

- Pros:
  - rich, dynamic notion of context that tracks some of the *history* of the computation.
  - *this* could stop Melissa, Love Bug, etc.
  - low overhead, no real state needed.
- Cons:
  - implementation-driven (walking up stacks)
    - Could be checked statically [Wallach]
  - policy is smeared over program
  - possible to code around the limited history
    - e.g., by having applets return objects that are invoked after the applet's frames are popped.
  - danger of over/under amplification

Require type safety?

- Write all security-critical programs in type-safe high-level language? (e.g., Java)
- Problem 1: legacy code written in C, C++
  - Solution: type-safe, backwards compatible C
- Problem 2: sometimes need control over memory management
  - Solution: type-safe memory management
- Can we have compatibility, type safety *and* low-level control? Can get 2 out of 3:
  - CCured [Necula et al. 2002]
    - Emphasis on compatibility, memory safety
  - Cyclone [Jim et al. 2002]
    - Emphasis on low-level control, type safety

Cyclone

- A type-safe dialect of C
- Goals:
  - Memory and type safety (*fail-stop* behavior)
  - (relatively) painless porting from C
  - writing new code pleasant
  - Low-level control: data representations, memory management, ability to interface to the outside world, performance, *etc.*
- Has been used to implement low-level, code safely, e.g. device drivers

Hello World

```
#include "stdio.h"

int main(int argc, char ??argv) {
  if (argc < 1) {
    fprintf(stderr,"usage: %s <name>\n",argv[0]);
    exit(-1);
  }
  printf("Hello, %s\n",*(++argv));
  return 0;
}

% a.out 'World!'
Hello World!
```


The trouble with pointers

- Pointer arithmetic:

```
int *f(int *a) {
   return a + 10;
}
```

- Null pointers:

```
int *f(int *a) {
   return a[4];
}
```

- Arrays:

```
struct foo {
  int g[10];
}
int *f(struct foo *x)
{ return &x->g[5]; }
```

- The stack:

```
int x;
scanf("%d", &x);
```

- All possibly legitimate uses of C pointers
- How can compiler check them (and modularly)?

⇒Needs more information

Pointers

- Three kinds: fat, thin-null, thin-not-null

  You pay for what you get…

`char ?` : arbitrary pointer arithmetic, might be null, 3 words of storage, bounds checked

`char *` : no real arithmetic, might be null, 1 word of storage

`char @` : same as above, but never null

`char *{42}` : same, but points to (at least) 42 chars

`char * == char *{1}`

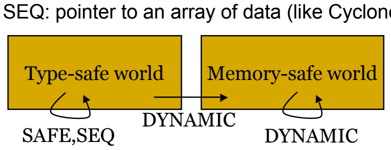
`char @{n+m} ≤ char @{n} ≤ char *{n} ≤ char ?`

Compatibility

- Porting most C code is fairly straightforward:
  - mostly, convert `t*` to `t?` where necessary
  - use advanced type features (polymorphism, tagged unions, existential types) to replace unsafe casts with type-safe operations
  - put in initializers (only top-level, interprocedural)
  - put in fallthru's (very rare)

CCured [Necula, 2002]

- Different pointer classes
  - DYNAMIC : no info, slow, all accesses checked
  - SAFE: a memory- and type-safe pointer (or null)
  - SEQ: pointer to an array of data (like Cyclone fat)



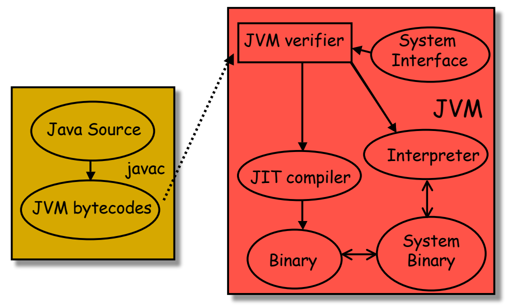
- Nonmodular but fast C→CCured converter using BANE constraint solving framework (worst case: DYNAMIC)
- 10-50% Performance penalty
- More safe C impls: [Jones&Kelly], [Ruwase&Lam]

Certifying compilation

Code certification mechanisms

- Problem: can you trust the code you run?
- Code signing using digital signatures
  - Too many signers
  - If you can't trust Microsoft,…
- Idea: self-certifying code
  - Code consumer can check the code itself to ensure it's safe
  - Code includes annotations to make this feasible
  - Checking annotations easier than producing them
- *Certifying compiler* generates self-certifying code
  - Java/JVM: first real demonstration of idea

Type-Based Protection (JVM)



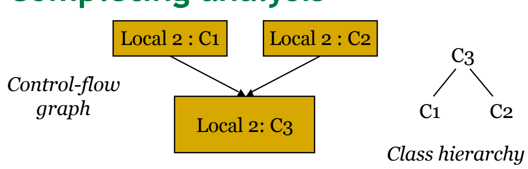
Bytecode verification

- Java compiler is a *certifying compiler* that compiles to Java Virtual Machine code
  - Generates enough type information in target code to check that code is type-safe
  - Same thing could be done with other source languages
  - Microsoft CLR is similar
- Verifier first checks structure (syntax) of bytecode
- Branches checked to ensure they address valid target instructions (*control safety*)
- Methods (functions) and class fields are annotated with complete type signatures (argument and result types)
- Method calls are explicit in JVM -- can look up signatures directly

Type-checking JVM

- Calls can be type-checked once actual argument types are known
- Java Virtual Machine stores data in locals (used for variables) and stack locations (used for arguments, temporaries)
  - Types of both can change at every program point, not included in bytecode format
- Verification uses dataflow analysis to determine types of every local/stack locn at every program point
- Use argument types and method result types to get analysis started

Completing analysis

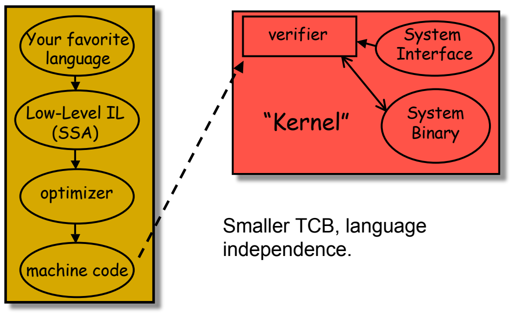


- Merge type information on different paths by finding least common ancestor (C3)
- If no least common ancestor mark type as unusable (local 2: ?)
- Report success if all method calls, bytecode operations type-check, otherwise reject program

Compiling to the JVM

- The JVM type-system isn't all that different from Java's ➔ compiling other languages to JVM doesn't work that well.
  - e.g., no tail-calls in the JVM so Scheme and ML are hosed... (MS fixed this in CLR)
  - no parametric polymorphism, no F-bounded subtyping, limited modules, etc.
- Operations of the JVM are relatively high-level, CISC-like.
  - method call/return are primitives
  - Little control over indirection
  - interpreter or JIT is necessary

Ideally:



Smaller TCB, language independence.

Typed Assembly Language

[Morrisett, 1998]

Two goals:

- Get rid of the need for a trusted interpreter or JIT compiler
  - type-check the actual code that will run.
  - try not to interfere with traditional optimizations.
- Provide generic *type constructors* for encoding many high-level language type systems.
  - reasonable target for compiling many languages
  - a more "RISC" philosophy at the type-level
  - better understanding of inter-language relationships.

TAL contributions

Theory:

- simple MIPS-like assembly language
- compiler from ML-like language to TAL
- soundness and preservation theorems

Practice:

- most of IA32 (32-bit Intel x86)
- more type constructors (array,+,μ,modules)
- prototype Scheme, Safe-C compilers

TAL (simplified)

Registers: $r \in \{$`r1`,`r2`,`r3`,...$\}$

Labels: $L \in \mathit{Identifier}$

Integer: $n \in [-2^{k-1}..2^{k-1})$

Blocks: $B ::=$ `jmp` $v \mid \iota\ ; B$

Instrs: $\iota ::= \mathit{aop}\ r_d, r_s, v \mid \mathit{bop}\ r, v \mid$ `mov` r, v

Operands: $v ::= r \mid n \mid L$

Arithmetic Ops: $\mathit{aop} ::=$ `add` | `sub` | `mul` | ...

Branch Ops: $\mathit{bop} ::=$ `beq` | `bne` | `bgt` | `bge` | ...

Simple program

```
; fact(n,a) = if (n <= 0) then a else fact(n-1,a*n)
;   r1 holds n, r2 holds a, r31 holds return address
;   which expects the result in r1
fact:    sub r3,r1,1  ; r3 := n-1
       ble r3,L2        ; if n < 1 goto L2
       mul r2,r2,r1     ; a := a*n
       mov r1,r3        ; n := n-1
       jmp fact         ; goto fact
L2:    mov r1,r2        ; result := a
       jmp r31      ; jump to return address
```


Basic type structure

$\mathit{type} ::= \mathrm{int} \mid \Gamma$

where $\Gamma = \{\ r_1{:}t_1,\ r_2{:}t_2,\ \ r_3{:}t_3,\ ...\}$

A value with type $\{\ \mathrm{r1} : t_1, \mathrm{r2} : t_2, \mathrm{r3} : t_3, ...\}$ is a code label, which when you jump to it, expects you to at least have values of the appropriate types in the corresponding registers.

You can think of a label as a function that takes a record of arguments

- the function never returns – it always jumps off
- we assume record subtyping – we can pass a label more arguments than it needs

Simple program with types

```
fact:{r1:int,r2:int,r31:{r1:int}}
     ; r1 = n, r2 = accum, r31 = return address
     sub  r3, r1, 1  ; {r1:int,r2:int,r31:{r1:int},r3:int}
     ble  r3, L2
     mul  r2, r2, r1
     mov  r1, r3
     jmp  fact
L2:{r2:int, r31:{r1:int}}
     mov r1, r2
     jmp r31
```

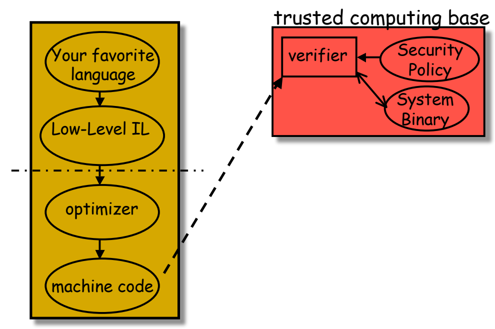

Badly typed program

```
fact:{r1:int,r31:{r1:int}}
     ; r1 = n, r2 = accum, r31 = return address
     sub  r3, r1, 1  ; {r1:int,r31:{r1:int},r3:int}
     bge  r1, L2
     mul  r2, r2, r1       ; ERROR!  r2 doesn't
  have a type
     mov  r1, r3
     jmp  L1
L2:{r2:int, r31:{r1:int}}
     mov r1, r2
     jmp r1       ; ERROR!  r1 isn't a valid label
```

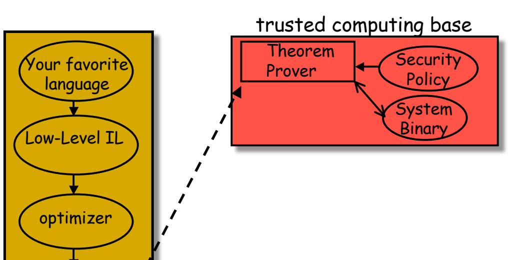

TAL vs JVM, CLR

- The principles behind TAL and the JVM (or Microsoft's CLR) aren't too different: compiler generates enough type annotations to check target code
- TAL concentrates on orthogonal, expressive typing components (more general target lang)
- JVM (and CLR) focus on OO-based languages with predefined implementation strategies.

Ideally:

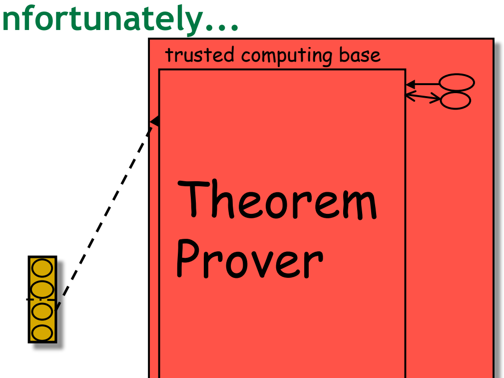


Idea #1: Theorem Prover!



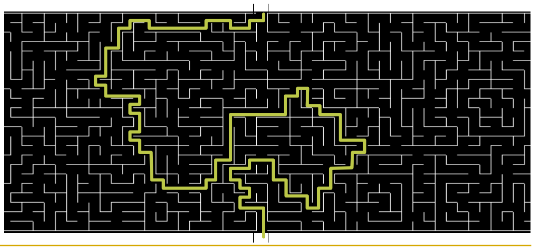
Warning: components not drawn to scale!

Unfortunately...

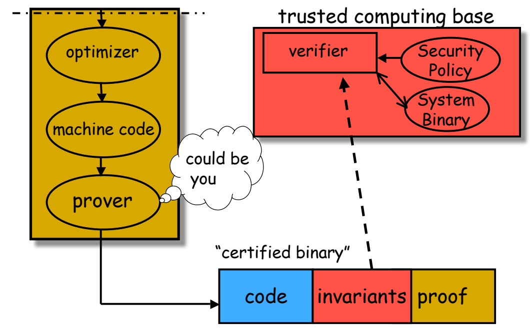


Observation

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



PCC:



Making "Proof" Rigorous:

Specify machine-code semantics and security policy using axiomatic semantics.

{Pre} `ld r2,r1(i)` {Post}

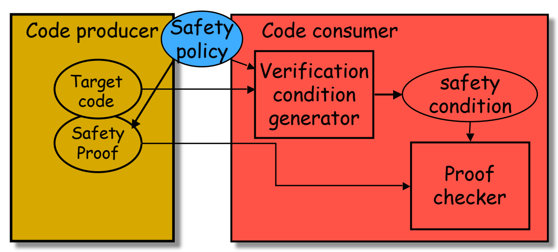
Given:

- security policy (*i.e.*, axiomatic semantics and associated logic for assertions)
- untrusted code, annotated with (loop) invariants

it's possible to calculate a *verification condition:*

- an assertion A such that
- if A is true then the code respects the policy.

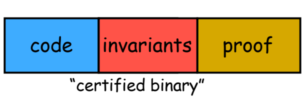
Proof-carrying code



Code consumer side

Verifier (~5 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

A generic architecture for providing and checking safety properties

In Principle:

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

In Practice:

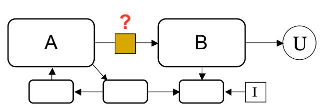
- Still hard to generate proofs for properties stronger than type safety. Need certifying compiler…

Security types and information flow

End-to-end security

- Near-term problem: ensuring programs are memory-safe, type-safe so fine-grained access control policies can be enforced
- Long-term problem: ensuring that complex (distributed) computing systems enforce system-wide information security policies
  - Confidentiality
  - Integrity
  - Availability
- Confidentiality, integrity: end-to-end security described by *information-flow policies* that control *information dependency*

Policies vs. mechanisms



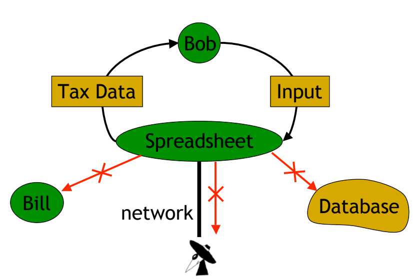
- Problem: policy/mechanism mismatch
  - Reference monitors (e.g., access control): control whether A is allowed to transmit to B
  - Confidentiality policy: information I can only be obtained by users U (no matter how it is transformed) – not a safety policy!
- How to map policy onto a mechanism? (we already do this by hand!)

Problems

- Complex policies
  - no generally accepted policy language
  - weak validation techniques
- Information flows hard to find (*covert channels*)
- Heterogeneous, changing trust
- Host machines may be compromised

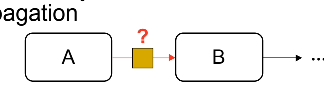
Information leaks

- Programs can leak inputs

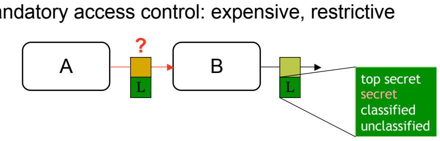


Standard mechanisms

- Discretionary access control: no control of propagation



Mandatory access control: expensive, restrictive

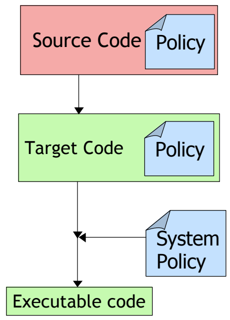


Java stack inspection: integrity, not confidentiality
Can't enforce information flow policies

Static information flow

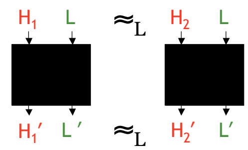
[Denning & Denning, 1977]

- Programs are annotated with information flow policies for confidentiality, integrity
- Compiler checks, possibly transforms program to ensure that all executions obey rules
- Loader, run-time validates program policy against system policies



Noninterference

"Low-security behavior of the program is not affected by any high-security data."
Goguen & Meseguer 1982



Confidentiality: high = confidential, low = public

Integrity: low = trusted, high = untrusted

Security types

- Idea: add information flow policies as type annotations (*labels*)
- Simplest policy language: H = confidential, L = public. L →H ok, H→L bad

```
int{H} x; // confidential integer
int{L} y; // public integer
String{L} z; // public string
x = y; // OK
y = x; // BAD
x = z.size(); // OK
z = Integer.toString(x) // BAD
```


Lattices and security

- Denning, 1976.
- Information flow policies (security policies in general) are naturally *partial orders*
  - If policy P_2 is stronger than P_1, write $P_1 \sqsubseteq P_2$
    - P1 = "smoking is forbidden in restaurants"
    - P2 = "smoking is forbidden in public places"
  - Some policies are incomparable: $P_1 \not\sqsubseteq P_2$ and $P_2 \not\sqsubseteq P_1$
    - P2 = "keep off the grass"
- If there is always a least restrictive policy as least as strong as any two policies, policies form *lattice*. $P_1 \sqcup P_2$ = "join" of P_1, P_2
  - $P_1 \sqcup P_2$ = "smoking forbidden in restaurants and keep off the grass"
- $H \sqcup H = H$, $L \sqcup L = L$, $L \sqcup H = H$

Checking computation

- Combining values with different information flow policies?
- Conservatively,

  Label of result should be a policy at least as strong as the labels of all inputs.
- Write x̲ for "label of **x**"
- Label of **y+z** is y̲ ⊔ z̲

Implicit Flows

```
x = 0;
if (b) {
  x = a;
}
```

- Final value of **x** may reveal values of **a**, **b**
- Conservative: label of **x** protects both **a** and **b**

a̲ ⊑ x̲ & b̲ ⊑ x̲

Static Assignment Rule

- Program-counter label *pc* captures implicit flows
- `if`, `while`, `switch` statements bump up *pc* (temporarily)

```
x = 0;
if (b) {
   x = a;     ← pc = b
}
```

Compile-time checking:

$a \sqsubseteq x$

$pc \sqsubseteq x$

Run-time Checking?

```
x = 0;
if (b) {
   x = a;     ← a ⊑ x & b ⊑ x ?
}
```

- Run-time check
  - if b is false, x=0, but no check is performed
  - if check fails, b is also leaked!
- Static checking not just an optimization!

Proving noninterference

- Volpano et al., 1996
- Can show that any type-safe program with information-flow security types must satisfy noninterference
- Strategy: show that each step of execution preserves low-observable equivalence:

$$\begin{array}{ccc} P_1 & \approx_L & P_2 \\ \downarrow & & \downarrow \\ P_1' & \approx_L & P_2' \\ \downarrow & & \downarrow \\ \ldots & & \ldots \end{array}$$

- Language with functions, state: Zdancewic, Myers '01
- Core ML: Pottier, 2002

Jif: Java + Information Flow

[Myers, 1999]

- Annotate (Java) programs with labels from decentralized label model
- Variables have type + label. Labels contain policies in terms of principals.

  `int {Alice→Bob} x;`
- Information flow control **and** access control

```
float {x} cos (float x) {
   float {x} y = x – 2*PI*(int)(x/(2*PI));
   return 1 – y*y/2 + ...;
}
```

Available for download:
http://www.cs.cornell.edu/jif

Confidentiality policies as types

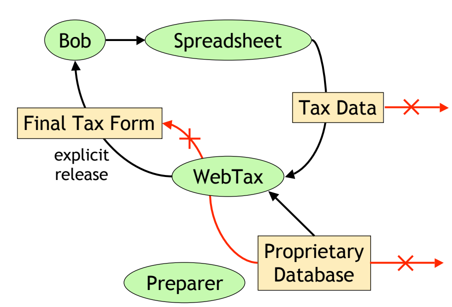
- Confidentiality labels:
  `int{Alice→} a;` **"a is Alice's private int"**
- Integrity labels:
  `int{Alice←} a;` **"a is trusted by Alice"**
- Combined labels:
  `int{Alice→ ; Alice←} a;` **(Both)**

```
int{Alice→} a1, a2;
int{Bob←} b;
int{Bob←Alice} c;
```

"Bob believes Alice can affect c"

// Insecure	// Secure
b = a1;	**a1 = a2;**
b = c;	**a1 = b;**
	a1 = c;
	c = b;

Intentional leaks



Selective Declassification

[Myers, 1997]

- An escape hatch from noninterference
- A principal can rewrite its part of the label

{O1→R1, R2; O2→R2}

O2

{O1→R1, R2}   {O1→R1, R2; O2→R2, R3}

- Other owners' policies still respected
- Must test authority of running process
- Potentially dangerous: explicit operation
- Jif 3.0: declassification mediated by integrity checks: "robustness" [Chong&Myers]

Information flow and dependency

- Checking whether information flows from x to y is just a dependency analysis
- Dependency is crucial to security!
- Many other applications of language-based dependency analysis…

Catching bad format strings

[Shankar et al., 2001]

- Idea: should not use untrustworthy (H) data as format string, else attacker can gain control of system
- Security type:
  `int printf(char *`$_L$` fmt, …)`
- Give network buffer type `char *`$_H$: information flow analysis prevents buffer data from affecting format string
  - problem: false positives
- Probably useful for less direct attacks too

SQL injection vulnerabilities

- WebSSARI system [Huang et al.], [Xie & Aiken]: analyze dependencies in PHP scripts to discover SQL queries built from untrusted information

```
$rows=mysql query("UPDATE users      SET pass='$pass'
WHERE userid='$userid'");
```

- $userid, $pass must be trusted information
- Sanitization functions convert untrusted to trusted after checking for metacharacters etc.
- Doesn't worry about implicit flows -- attacker can affect SQL queries but *probably* difficult to synthesize attacks…

Detecting worms

- Vigilante system [Costa et al.] uses dynamic and static dependency analysis to
  - Detect worm attacks
  - Automatically generate filters
  - Generalize filters so they catch larger class of related attacks
  - No false positives
- Filters can be distributed by peer-to-peer system in 2.5 min. (a solution to Slammer!)

Vigilante

- Idea: labels on data are sets of bytes from network messages where $\sqsubseteq$ is $\subseteq$
- Run app with dynamic binary rewriting, computing labels for data
- At invalid step (e.g., jumping to payload), label on step says which message bytes matter!
- Generate filter from them.

Other work and future challenges

- Security types for secrecy in network protocols
- Self-certifying low-level code for object-oriented languages
- Applying interesting policies to PCC/IRM
- Secure information flow in concurrent systems
- Enforcing availability policies

The End

- Thank you for your participation!
- See website for bibliography, more tutorial information: www.cs.cornell.edu/andru/pldi06-tutorial

Acknowledgements: Greg Morrisett, Fred Schneider, Steve Zdancewic, George Necula, Peter Lee