



Language-Based Security

Andrei Sabelfeld
Chalmers

<http://www.cs.chalmers.se/~andrei/>

Acknowledgements: David Sands,
Andrew C. Myers, and Steve Zdancewic

FOSAD, Sept. 2004

Course outline: the four hours

- today {
1. Language-Based Security: motivation
 2. Language-Based Security: the big picture
 3. Confidentiality for **sequential** and **multithreaded** programs (on the board)
 4. Mechanisms for safe **information release**
- + Extracurricular activities (find me after dinner)

A scenario: free service software

Users freely download and use the software providing a service:

- Grokster, Kazaa, Morpheus,... are file-sharing services helping users exchange files
- Come with “hooks” for automatic updates
- Support advertisement to justify cost



Real story: scumware

Users are tricked to download software bundled with **pests**:

- Homepage/search **hijackers** (MySearch)
- Unsolicited pop-up ads
- Rewriting URLs to override original ads with own
- “Hooks” for automatic updates are used to execute the advertiser’s **arbitrary code** (MediaUpdate, DownLoadware)
- Information gathering—visited URLs and filled forms are forwarded to a third-party (Gator, IPInsight, Transponder)



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



Types of malicious code 1

- **Viruses**: pieces of malicious code that attach to programs and propagate when an infected program executes
- **Worms**: carry out pre-programmed attacks spreading from machine to machine across network
- **Trojan** horses: malicious intent, yet appearing to do something useful (e.g., login daemon, web-spoofing)

Types of malicious code 2

- **Attack scripts**: written by experts to exploit security weaknesses (e.g., buffer overflow)
- **Java attack applets**: embedded in Web pages to achieve access through a Web browser
- **ActiveX controls**: program components that allow malicious code to control applications or the OS

Brief history of malicious code

- 1980's: Trojan hoarse, viruses (must be compact to keep to small volumes of the media)
- 1992: Web arrives
- 1995: Java and Javascript introduce widespread mobile code
- 1999: Melissa
- 2000: Love Bug (\$10bln damage)
- 2001: AnnaKournikova worm
- 2001: Code Red
- 2002: MS-SQL Slammer (published by MS)
- 2003: Blaster

Flexibility of Mobile Code

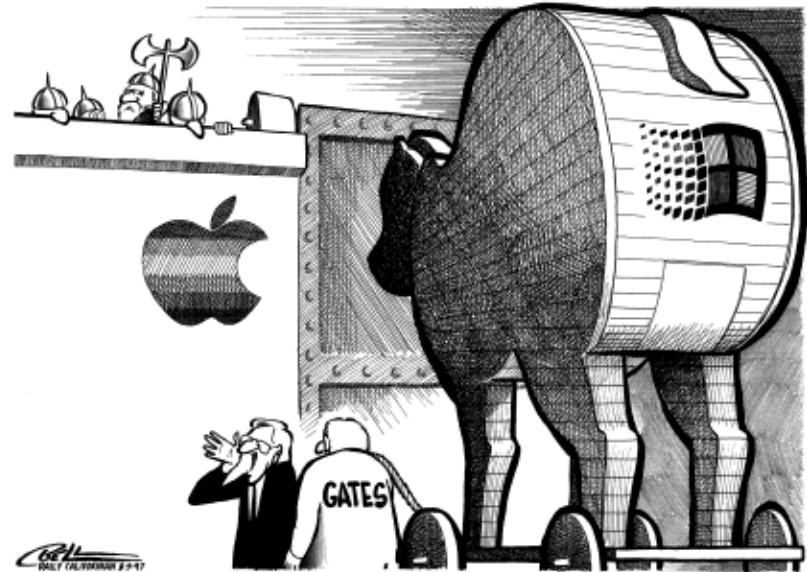
- Send around data that automatically executes
- The more platform the better
- Embedded, mobile devices need this

Examples are:

- Java, ActiveX, Postscript, Audio Codex, Word macros, JavaScript, VBS,...

Trojan Horses

- Any mobile program code may contain an (un)intended Trojan!



"He says he comes bearing gifts!"

Numerous Opportunities for the Attacker!

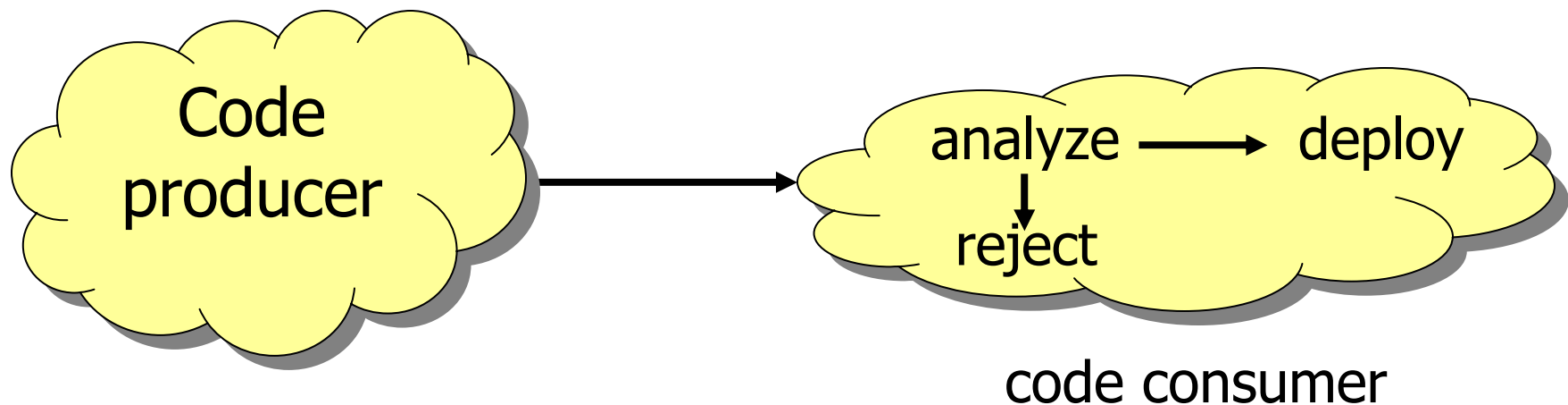
- JavaScript: invasion of privacy, denial of service, Web spoofing
- Macro pests: Melissa, Love Bug, AnnaKournikova worm
- ActiveX: system modification attacks, stealing money
- Java security: attack applets

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

Promising New Defenses via Language-Based Security 1

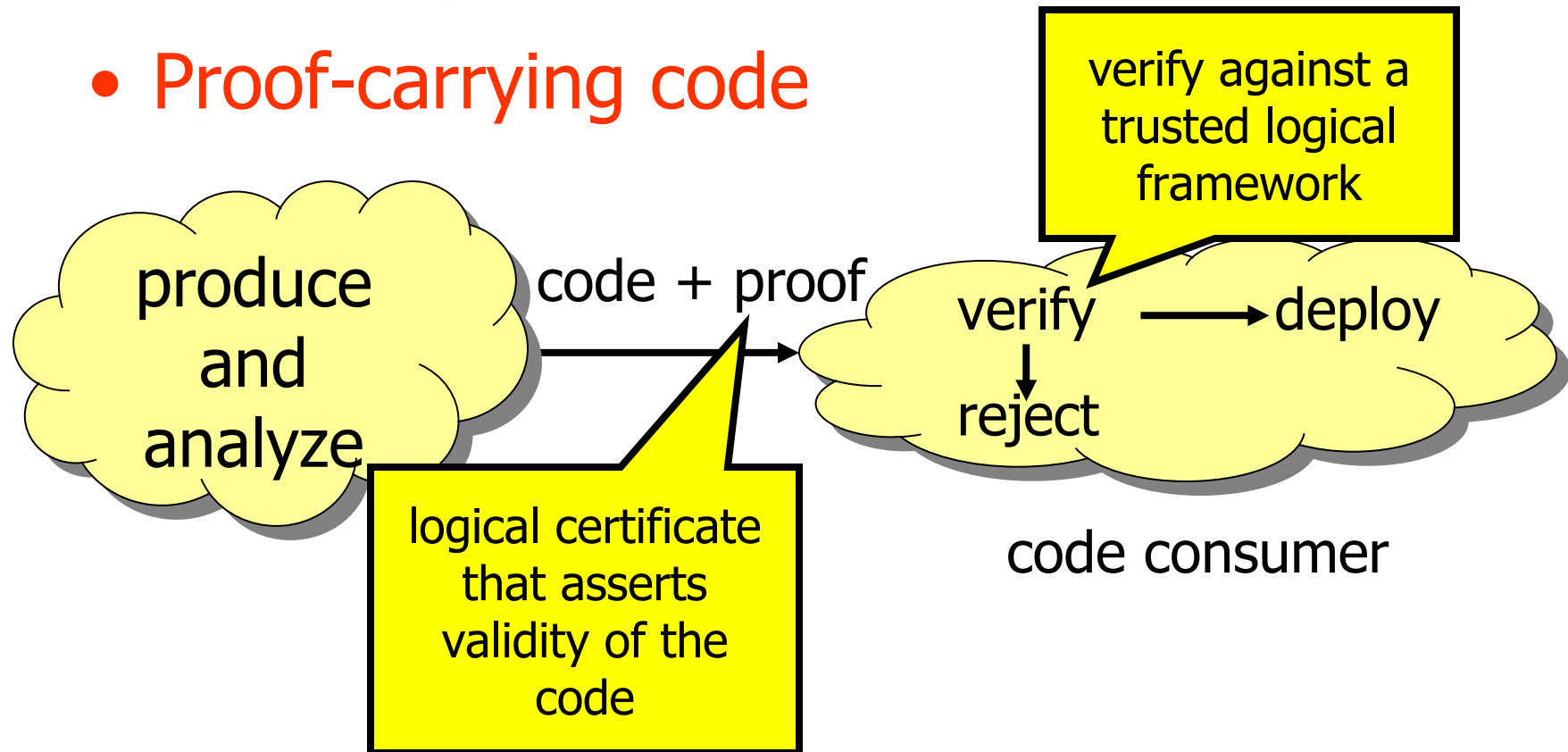
- Static certification e.g. type systems



- Main focus of the course

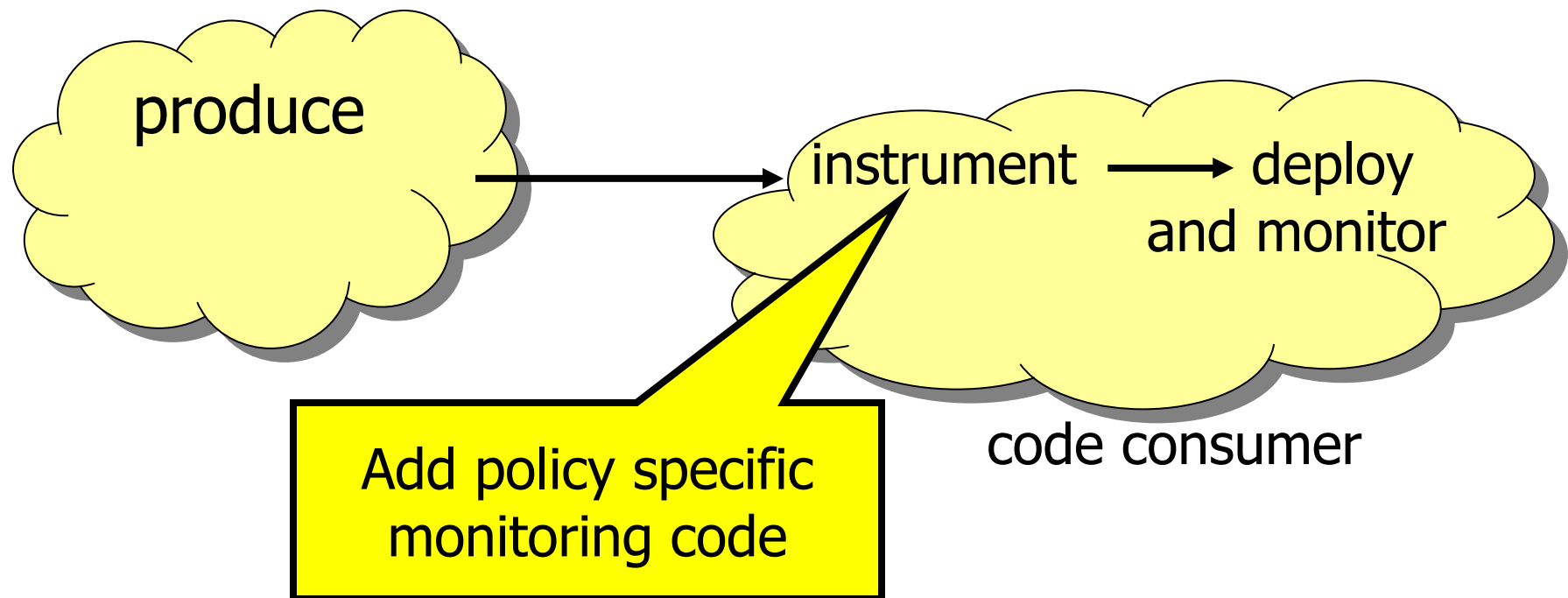
Promising New Defenses via Language-Based Security 2

- Proof-carrying code



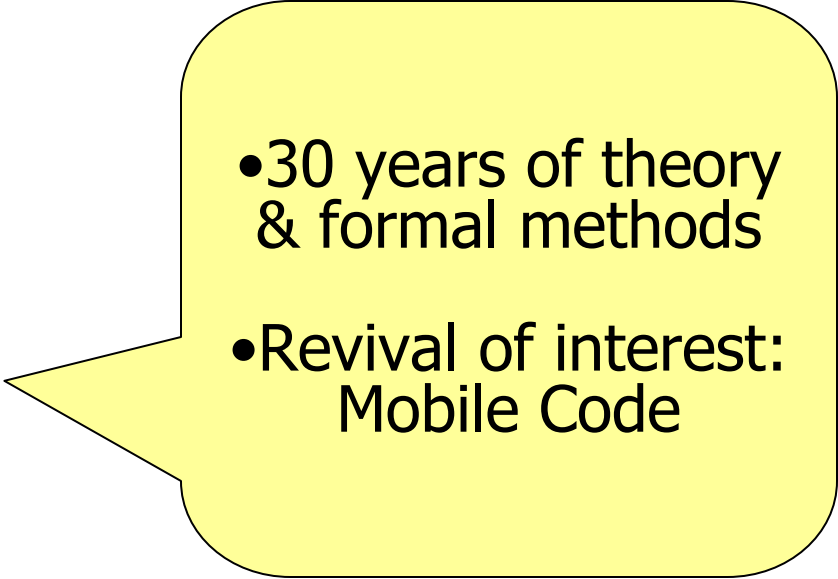
Promising New Defenses via Language-Based Security 3

- Software-based reference monitors

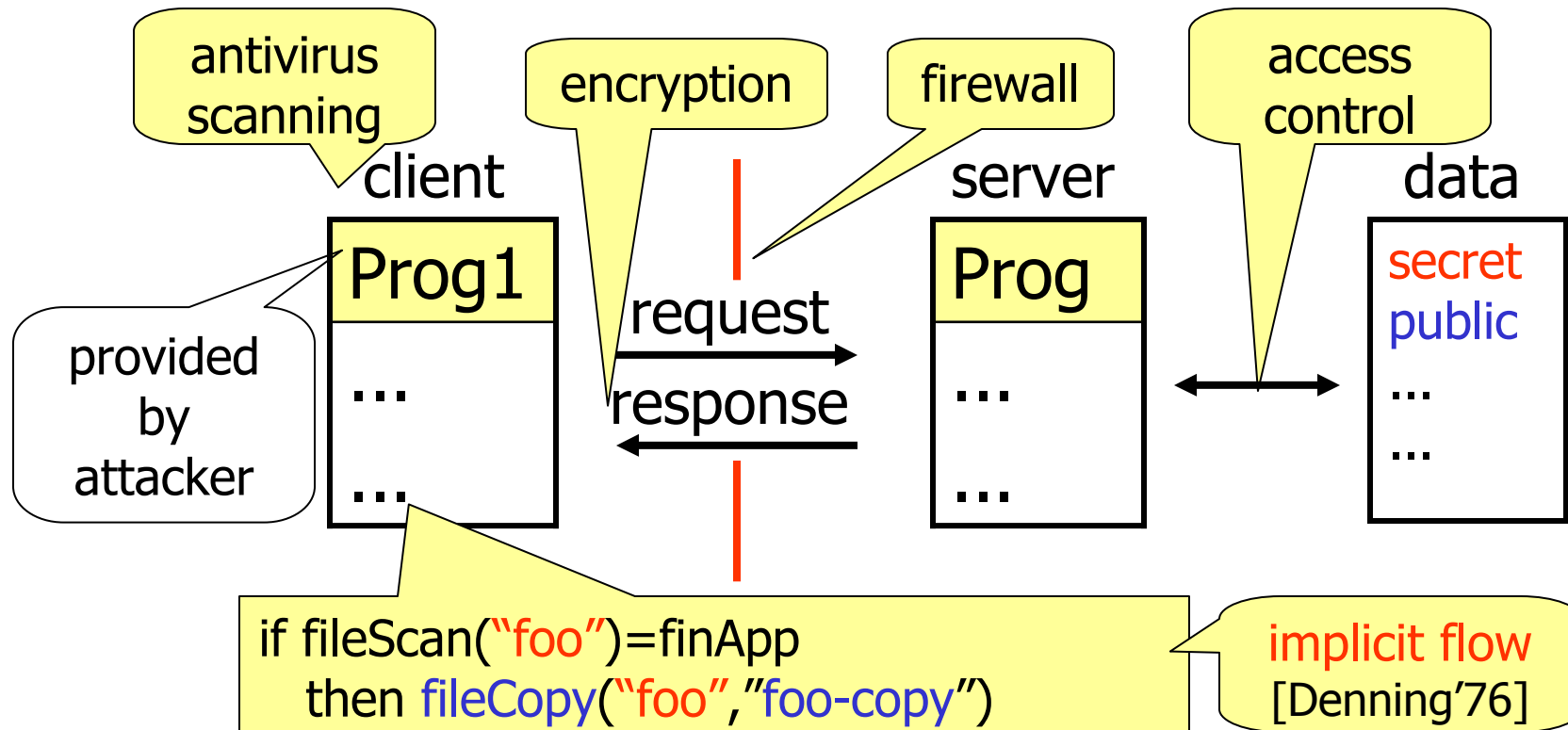


Computer Security

- The CIA
 - Confidentiality
 - Integrity
 - Availability

- 
- 30 years of theory & formal methods
 - Revival of interest: Mobile Code

Confidentiality: Motivation



- Distributed file client/server: attacker's goal is to learn **secrets** by observing **public** information
- To guarantee **end-to-end** security need **information-flow** controls

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

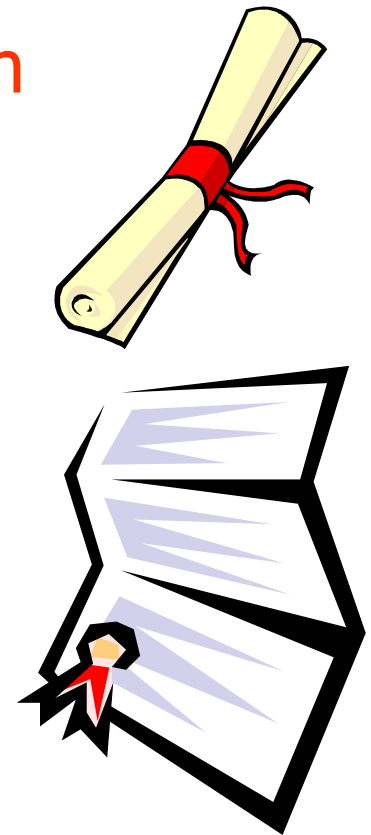
- +help identify code producer
- no security policy or security proof guaranteed

Sandboxing/OS-based monitoring

- +good for low-level events (such as read a file)
 - programs treated as black boxes
- ⇒ Useful building blocks but no **end-to-end** security guarantee

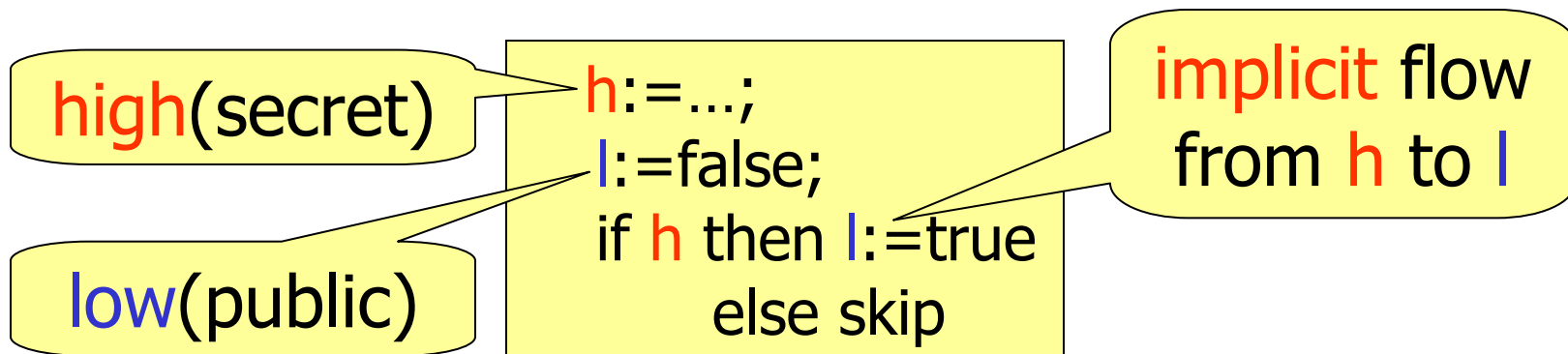
Confidentiality: language-based approach

- 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
- **Static security analysis**
 - Analysis enforcing end-to-end security
 - Track information flow via **security types**
 - Type checking by the compiler removes run-time overhead



Dynamic security enforcement

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



Problem: have to monitor **all** execution paths!

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]
- More precise implicit flow analysis
- Enforcement by **security-type systems**

A security-type system

Expressions:

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

$h \notin \text{Vars}(\text{exp})$

$\text{exp} : \text{low}$

Atomic commands (pc represents context):

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

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

$\text{exp} : \text{low}$

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

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

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

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

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

	<u>5 : low</u>	<u>3 : low</u>
[high] ⊢ h:=h+1	[low] ⊢ l:=5, [low] ⊢ l:=3, l=0: low	
[low] ⊢ h:=h+1	[low] ⊢ if l=0 then l:=5 else l:=3	
[low] ⊢ h:=h+1; if l=0 then l:=5 else l:=3		

What does the type system guarantee?

- Type soundness:

Soundness theorem:

$[pc] \vdash C \Rightarrow C \text{ is secure}$

what does it mean?

Semantics-based security

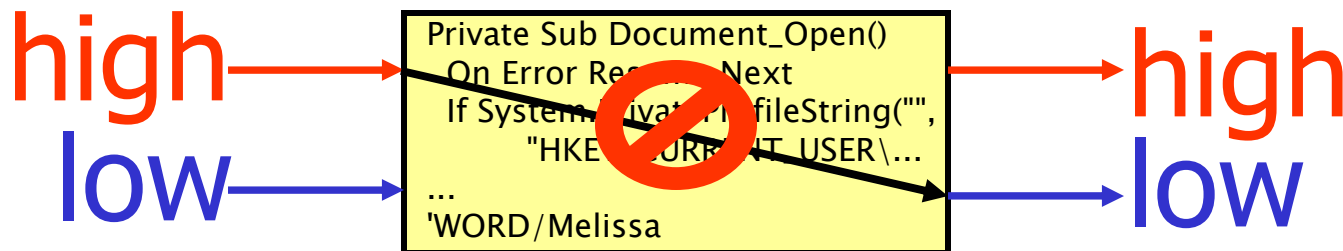
- What **end-to-end** policy such a type system guarantees (if any)?
- Semantics-based specification of information-flow security [Cohen'77], generally known as **noninterference** [Goguen&Meseguer'82]:

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

Confidentiality: assumptions (simplified)

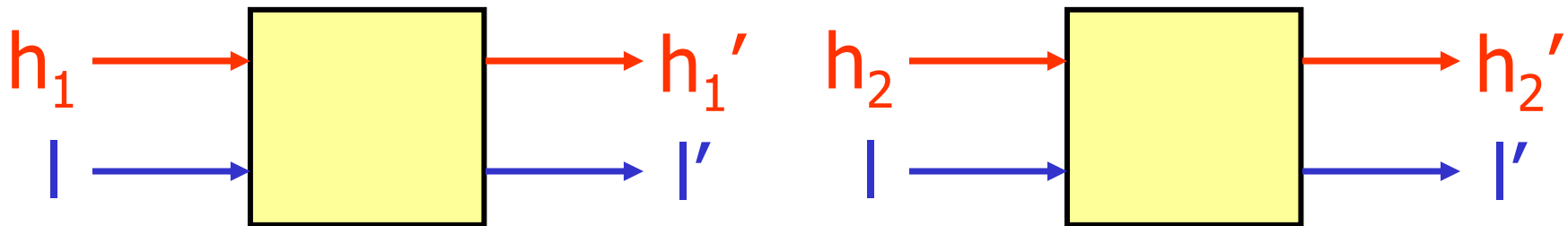
- Simple security structure (easy to generalize to arbitrary lattices)
- Variables partitioned: **high** and **low**

secret (high)
|
public (low)
- Intended security: **low**-level observations reveal nothing about **high**-level input:

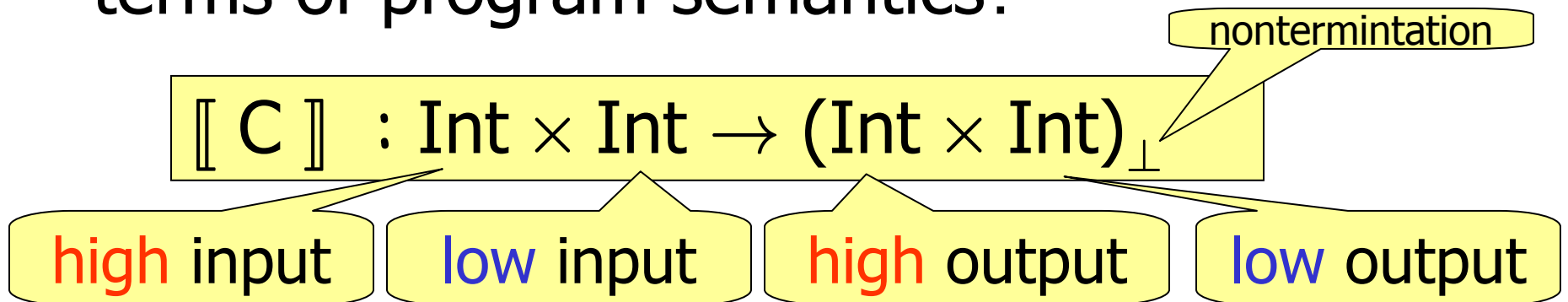


Confidentiality for sequential programs: noninterference

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



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



Semantics-based security

- Semantics-based security for C: as **high** input varied, **low**-level behavior unchanged:

$$\forall \text{mem}, \text{mem}'. \text{mem} =_L \text{mem}' \Rightarrow \llbracket C \rrbracket \text{mem} \approx_L \llbracket C \rrbracket \text{mem}'$$

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

C's behavior:
semantics $\llbracket C \rrbracket$

Low view $\approx_L$:
indistinguishability
by attacker

C is **secure** iff

$$\forall s_1, s_2. s_1 =_L s_2 \Rightarrow \llbracket C \rrbracket s_1 \approx_L \llbracket C \rrbracket s_2$$

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 $\approx_L$:

$$s \approx_L s' \text{ iff } s \neq \perp \neq s' \Rightarrow s =_L s'$$

Confidentiality: Examples

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

Semantics: Examples

$\llbracket l := h \rrbracket (x, y)$	(x, x)
$\llbracket l := h; l := 0 \rrbracket (x, y)$	$(x, 0)$
$\llbracket \text{if } h = 0 \text{ then } l := 0 \text{ else } l := 1 \rrbracket (1, y)$	$(1, 1)$
$\llbracket \text{while } h = 0 \text{ then do skip} \rrbracket (0, y)$	$\perp$
$\llbracket \text{if } h = 0 \text{ then sleep}(1000) \rrbracket (0, y)$	$(0, y)$

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]

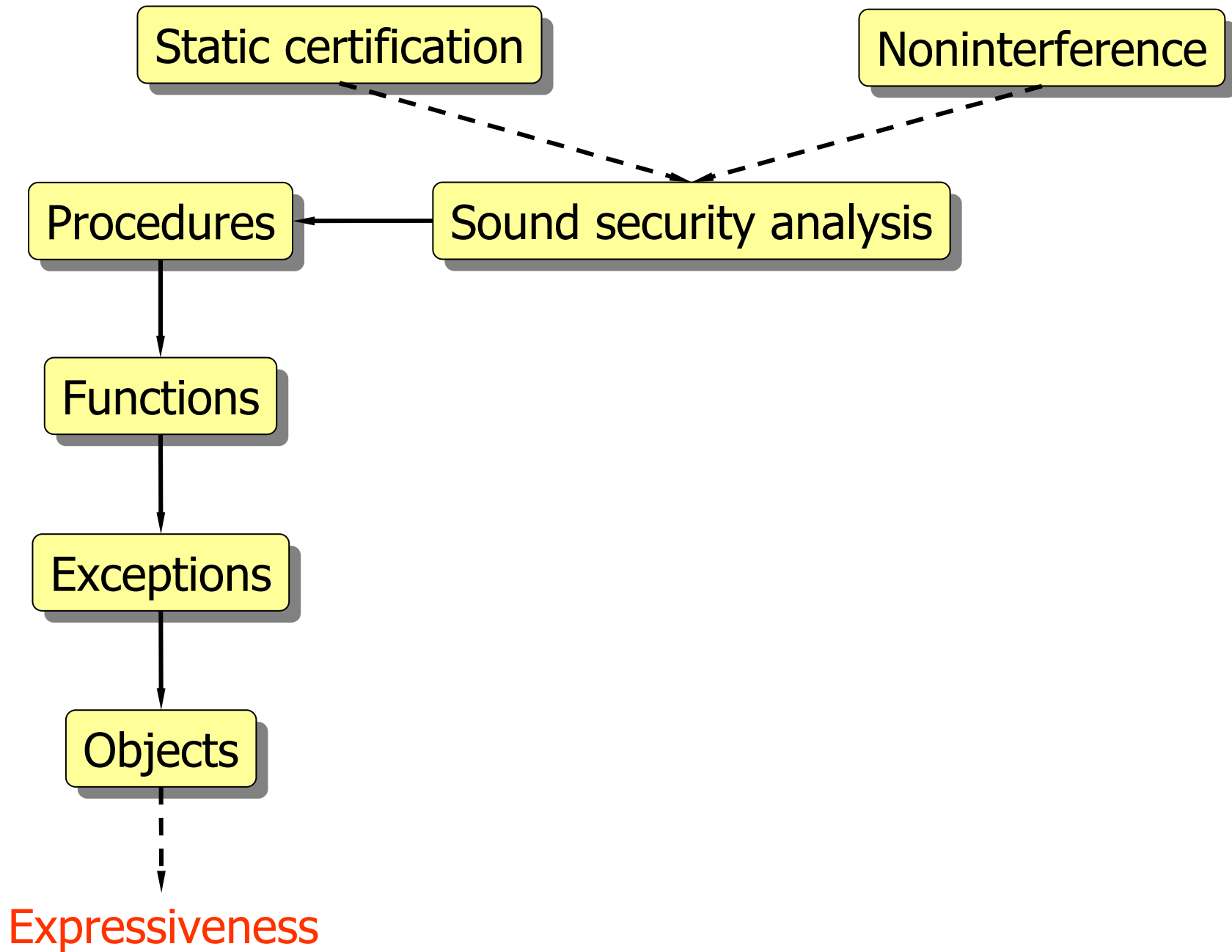
Security specification, e.g., [Cohen'77, Andrews& Reitman'80, Banâtre&Bryce'93, McLean'94]

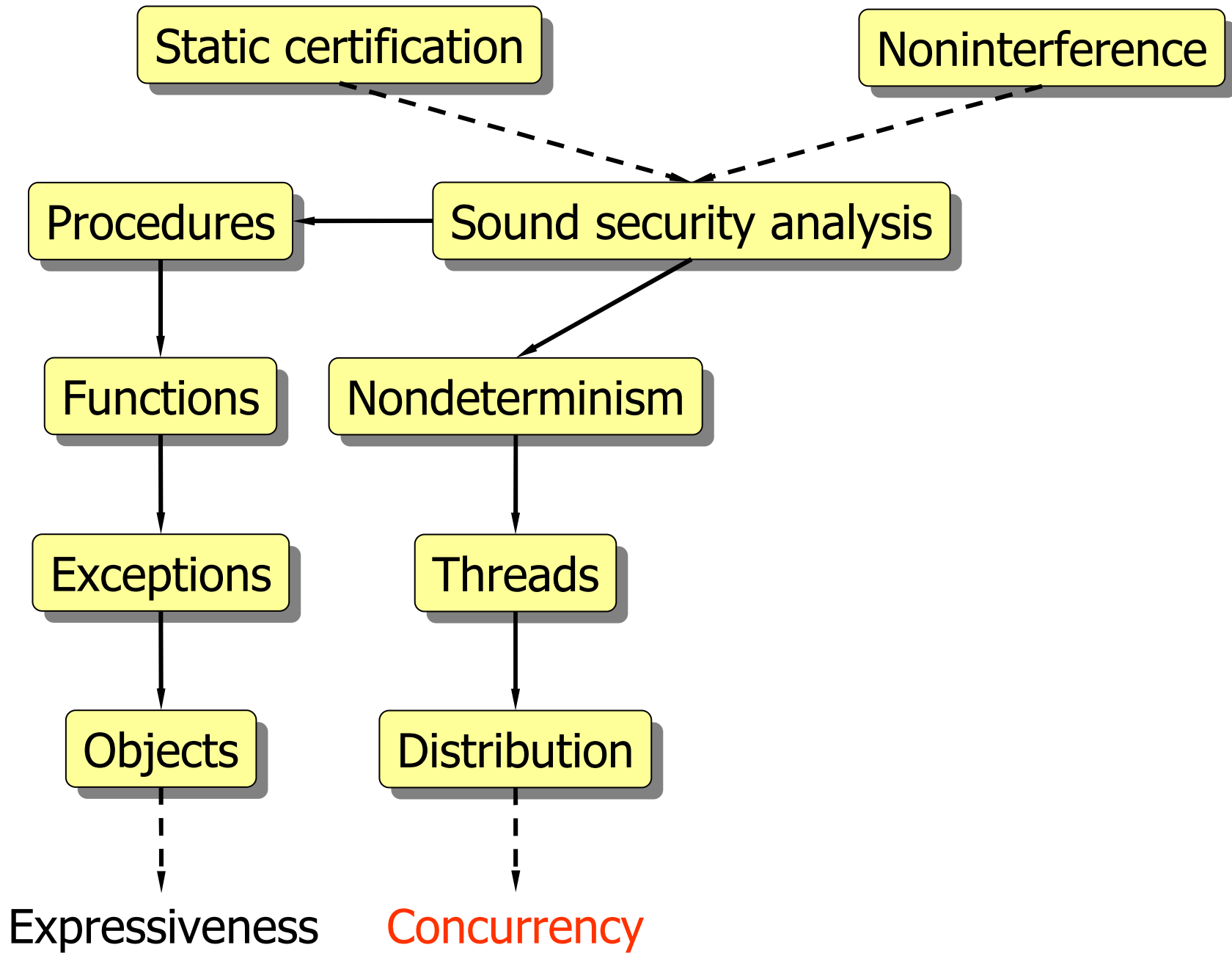
Volpano et al.'96: First connection between noninterference and static certification: security-type system that enforces noninterference

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**





Concurrency: Nondeterminism

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

$$\forall s. \llbracket HH; C; HH \rrbracket s \approx \llbracket C; HH \rrbracket s$$

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:

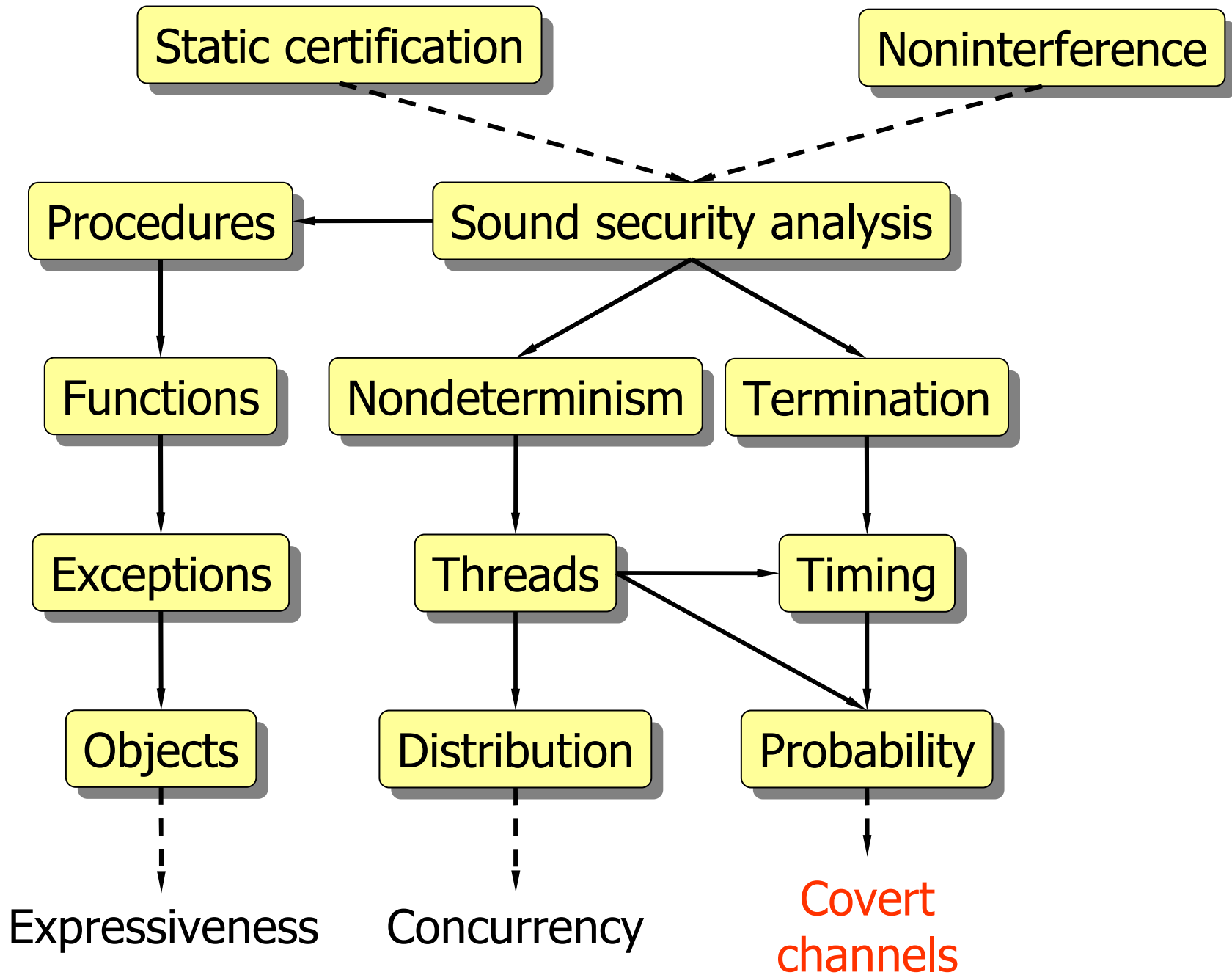
- Timing and prob. channels
- Compositionality
- Scheduler-independence
- Security type system

Concurrency: Distribution

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

Concurrency: Distribution

- An architecture for secure program splitting to run on heterogeneously trusted hosts [Zdancewic et al.'01, Zheng et al.'03]
- Type systems for secrecy for cryptographic protocols in spi-calculus [Abadi'97, Abadi&Blanchet'01]
- Logical relations for the low view [Sumii&Pierce'01]
- Interplay between communication primitives and types of channels [Sabelfeld&Mantel'02]



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):

$$s \approx_L s' \text{ iff } s = \perp = s' \vee (s \neq \perp \neq s' \wedge s =_L s')$$

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: $M^k \bmod n$

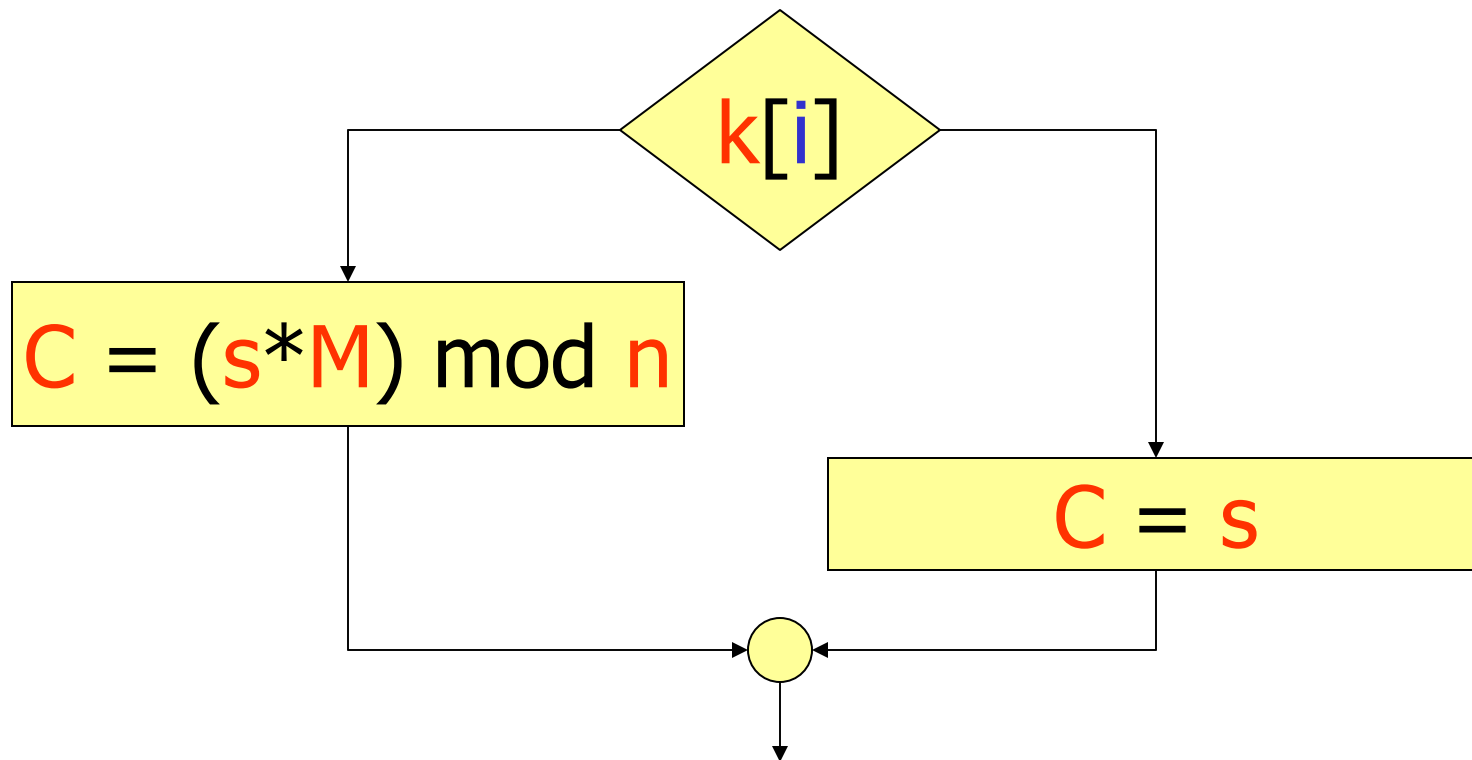
```
s = 1;
for (i=0; i<w; i++){
    if (k[i])
        C = (s*M) mod n;
    else
        C = s;
    s = C*C;
}
```

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

[Kocher'96]

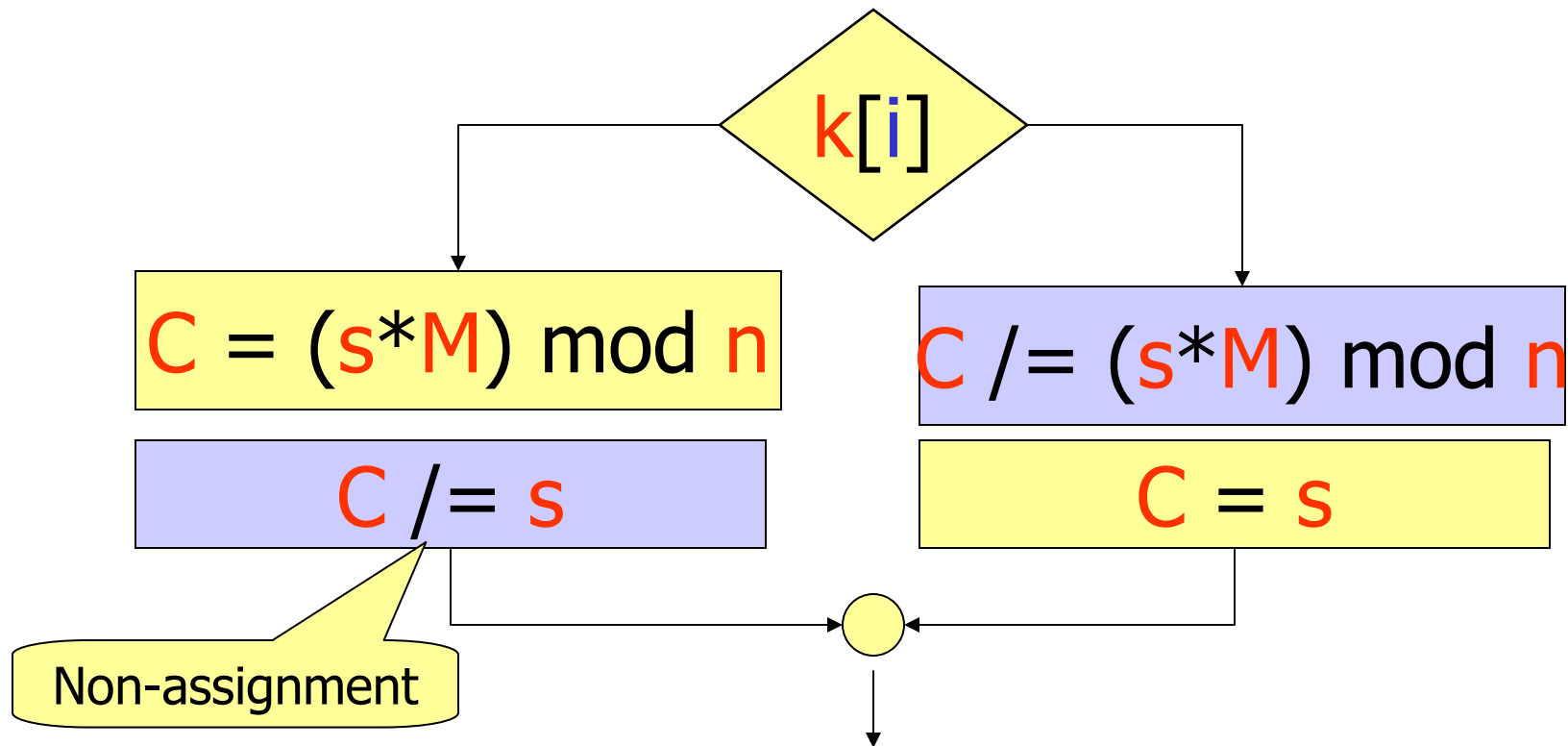
Transforming out timing leaks

Branching on **high** causes leaks



Transforming out timing leaks

Cross-copy **low slices**



Covert channels: Probabilistic

- Possibilistically but not probabilistically secure program:

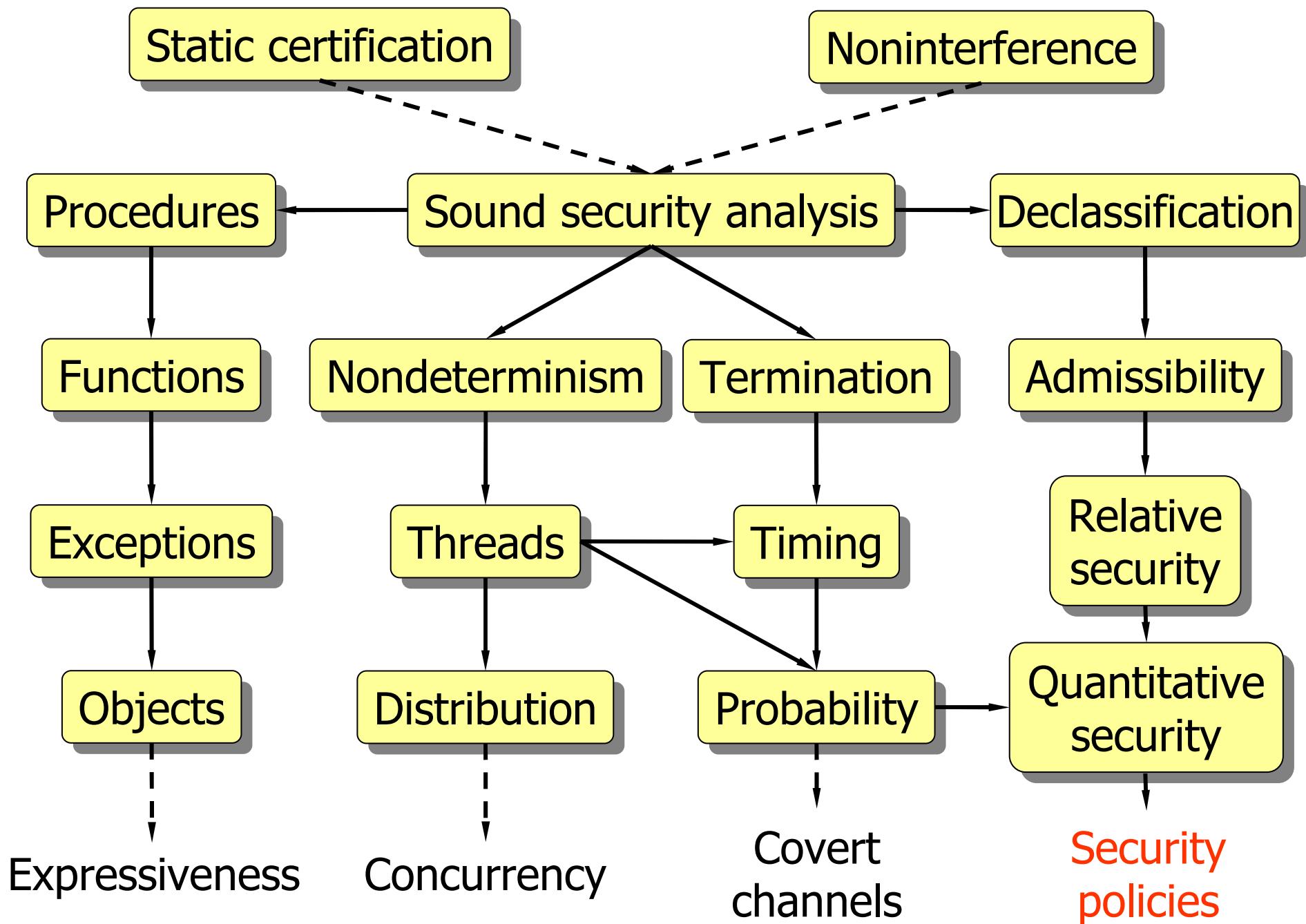
`l := PIN  $\square_{9/10}$  l := rand(9999)`

- Timing attack exploits probabilistic properties of the scheduler:

resolved by uniform scheduler

`(if h then sleep(1000)); l := 1 || 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]³



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
- Most approaches to information flow control ignore declassification—need more flexible security policies

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?

Security policies

- Secrecy in protocols [Abadi'97]
- Relative secrecy [Volpano&Smith'00, Volpano'00]
- Quantitative security [Denning'82, Clark et al.'02, Lowe'02]
- Approximate security $(\approx_L)_\varepsilon$ [Di Pierro et al.'02]
- Complexity-theoretic security [Laud'01,'03]
- Admissibility [Dam & Giambiagi'00, Giambiagi & Dam'03]
- Decentralized security model [Myers&Liskov'97]
- Robust declassification [Zdancewic&Myers'01, Zdancewic'03]
- Access control policies for secure information flow [Banerjee & Naumann'03]
- Cryptographic types [Duggan'02]
- Type-based distributed access control [Chothia et al.'03]

Language-based information security: challenges

Some essential challenges—some are not addressed by current trends!

- ▣ • System-wide security
- ▣ • Certifying compilation
- ▣ • Attacks beyond abstraction
- ▣ • Dynamic policies
- ▣ • Practical issues

⇒ Opportunities for integrating model checking, logic, theorem proving, code rewriting,...

Summary for today

- 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 via security type systems**)
- 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

Course outline: the four hours

1. Language-Based Security: motivation
 2. Language-Based Security: the big picture
 - tomor-
row { 3. Confidentiality for **sequential** and **multithreaded** programs (on the board)
 4. Mechanisms for safe **information release**
- + Extracurricular activities (find me after dinner)

References

- Language-based information-flow security [Sabelfeld & Myers, JSAC'03]
- Confidentiality for multithreaded programs via bisimulation [Sabelfeld, PSI'03]
- Static confidentiality enforcement for distributed programs [Sabelfeld & Mantel, SAS'02]
- Probabilistic noninterference for multithreaded programs [Sabelfeld & Sands, CSFW'00]
- A PER model of secure information flow in sequential programs [Sabelfeld & Sands, ESOP'99/HOSC'01]

End of talk



System-wide security

- Computer system only as secure as its weakest point
- Integration of language-based **local** computation and system-wide **global** information-flow controls [Mantel & Sabelfeld'01,'03]
- Secure program partitioning [Zdancewic et al.'01] and replication [Zheng et al.'03]: the security of the system as a whole
- Need rigorous connections to cryptographic protocols and trust management

Certifying compilation

- Much malicious code is distributed in low-level languages (such as applets, ActiveX)
- Information flow in low-level languages: information about control structure is lost; ordered linear continuations types enforce a stack discipline facilitating a high-precision analysis [Zdancewic & Myers'01]
- Certifying compilation (e.g., Java bytecode verification and typed assembly languages)

Abstraction-violating attacks

- Many attacks target circumventing the attacker model; e.g., **cache attack**

```
if h then h' := h1 else h' := h2;  
h' := h1
```

- An esoteric attack:

```
if h then mess = "hit enter"  
    else mess = "enter user id";  
input(mess, x)
```

Dynamic policies

- Common assumption on static label assignment
- Not realistic (e.g., access rights are dynamic)
- Dynamic security policies [Myers & Liskov'97] are implemented in Jif
- Dependent types complicate obtaining noninterference results

Practical issues

- Is it possible to write efficient secure programs?
- Basic algorithmic building blocks (**sorting** and **searching**) satisfy timing-sensitive noninterference without a substantial efficiency loss ($O(n \log n)$ and $O(\log n)$, resp.) [Agat & Sands'01]
- More experience on practical implications of secure information flow is needed