Information-flow security

Andrei Sabelfeld
Chalmers

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

GLOBAN, Sep. 2008

Course outline: the four hours

today

1. Language-Based Security: motivation
2. Language-Based Information-Flow Security: the big picture
3. Dimensions and principles of declassification
4. Combining the dimensions of declassification for dynamic languages

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
- 2005: Samy (MySpace worm, >1M pages)

Flexibility of Mobile Code

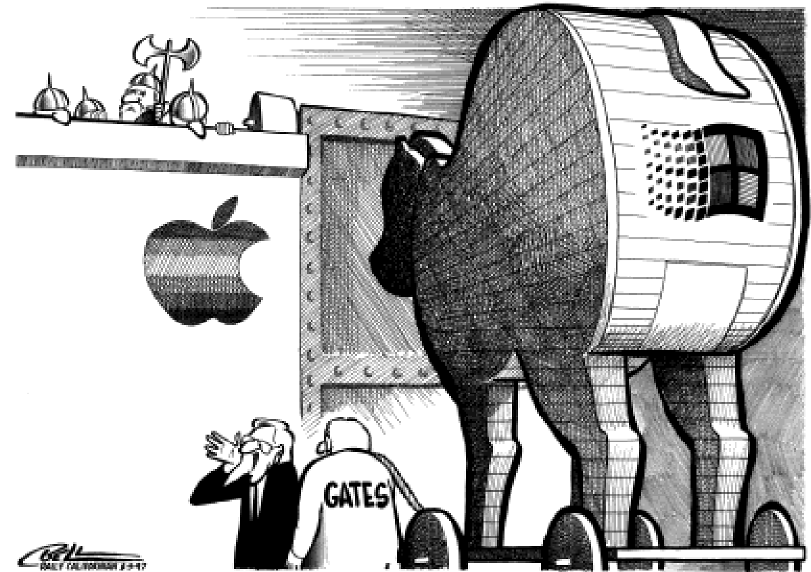
- Send around data that automatically executes
- The more platform the better
- Embedded, mobile devices need this

Examples are:

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

Trojan Horses

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



Numerous Opportunities for the Attacker!

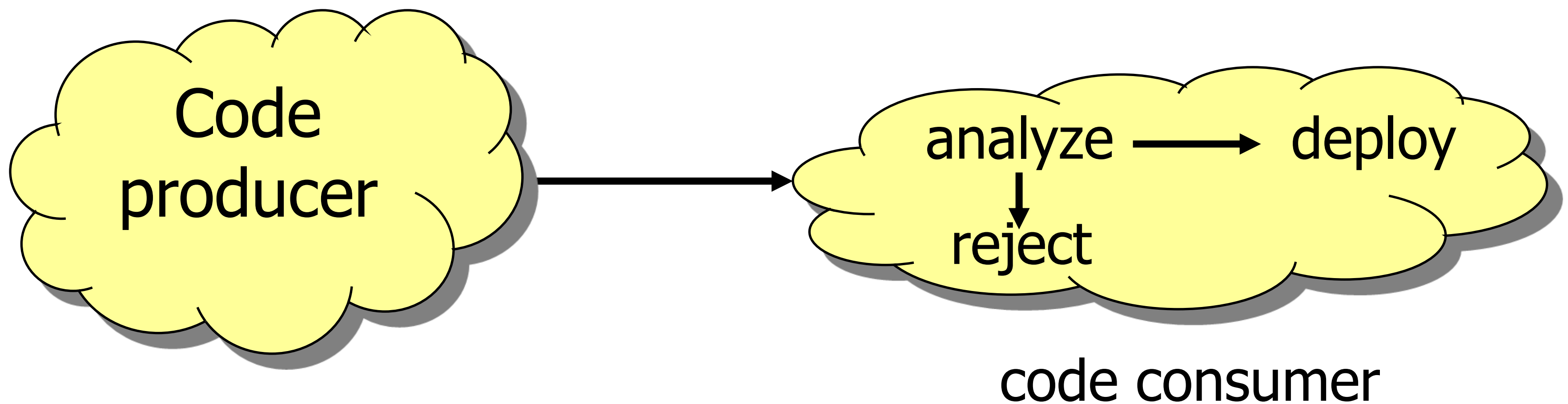
- JavaScript: invasion of privacy, denial of service, Web spoofing
- Macro pests: Melissa, Love Bug, AnnaKournikova worm
- ActiveX: system modification attacks, stealing money
- Java security: attack applets

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

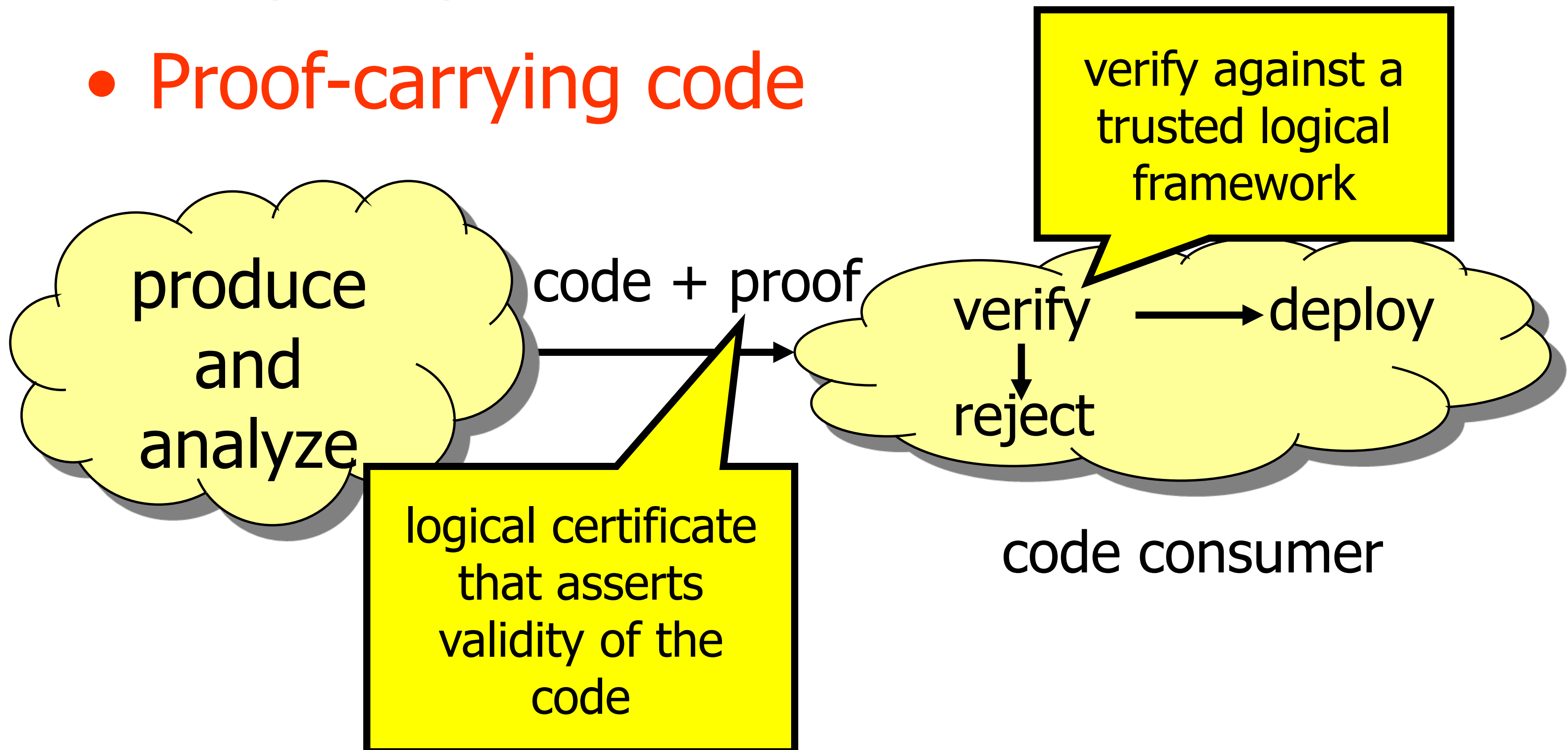
- Static certification e.g. type systems



- Main focus today

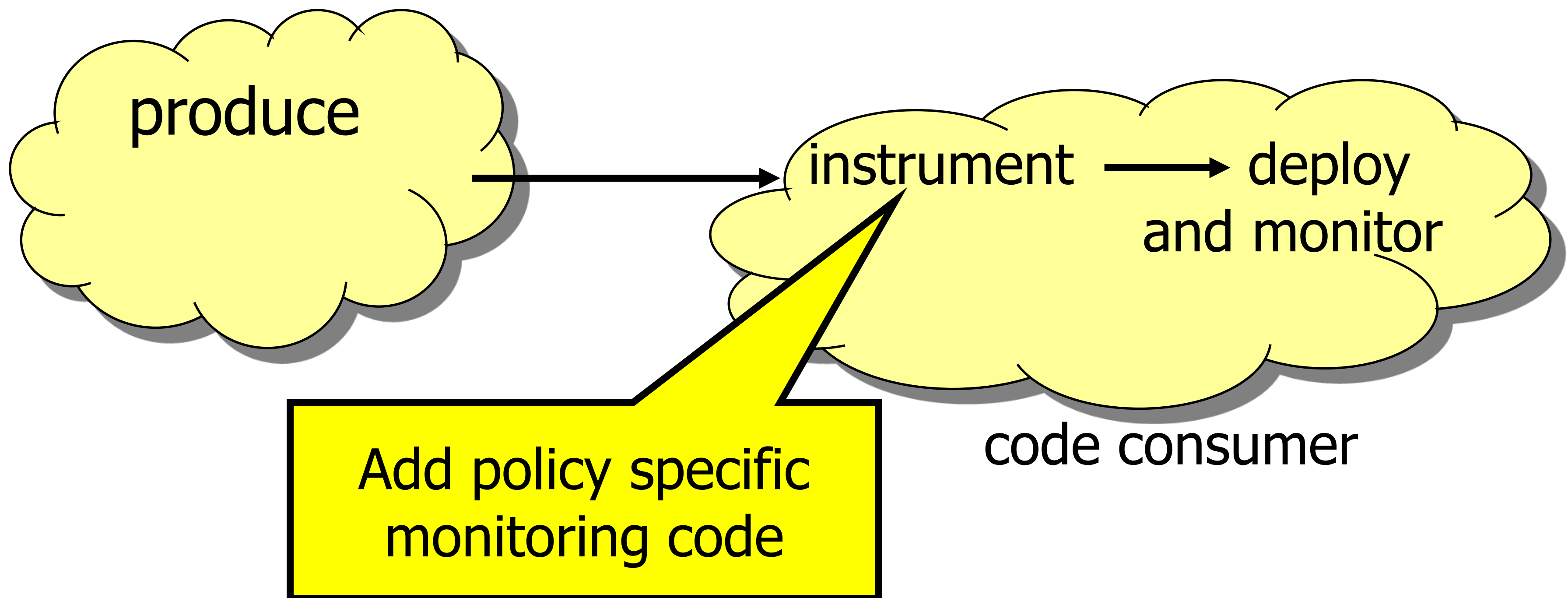
Promising New Defenses via Language-Based Security 2

- Proof-carrying code



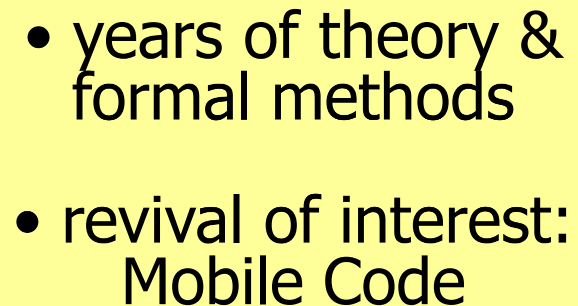
Promising New Defenses via Language-Based Security 3

- Software-based reference monitors

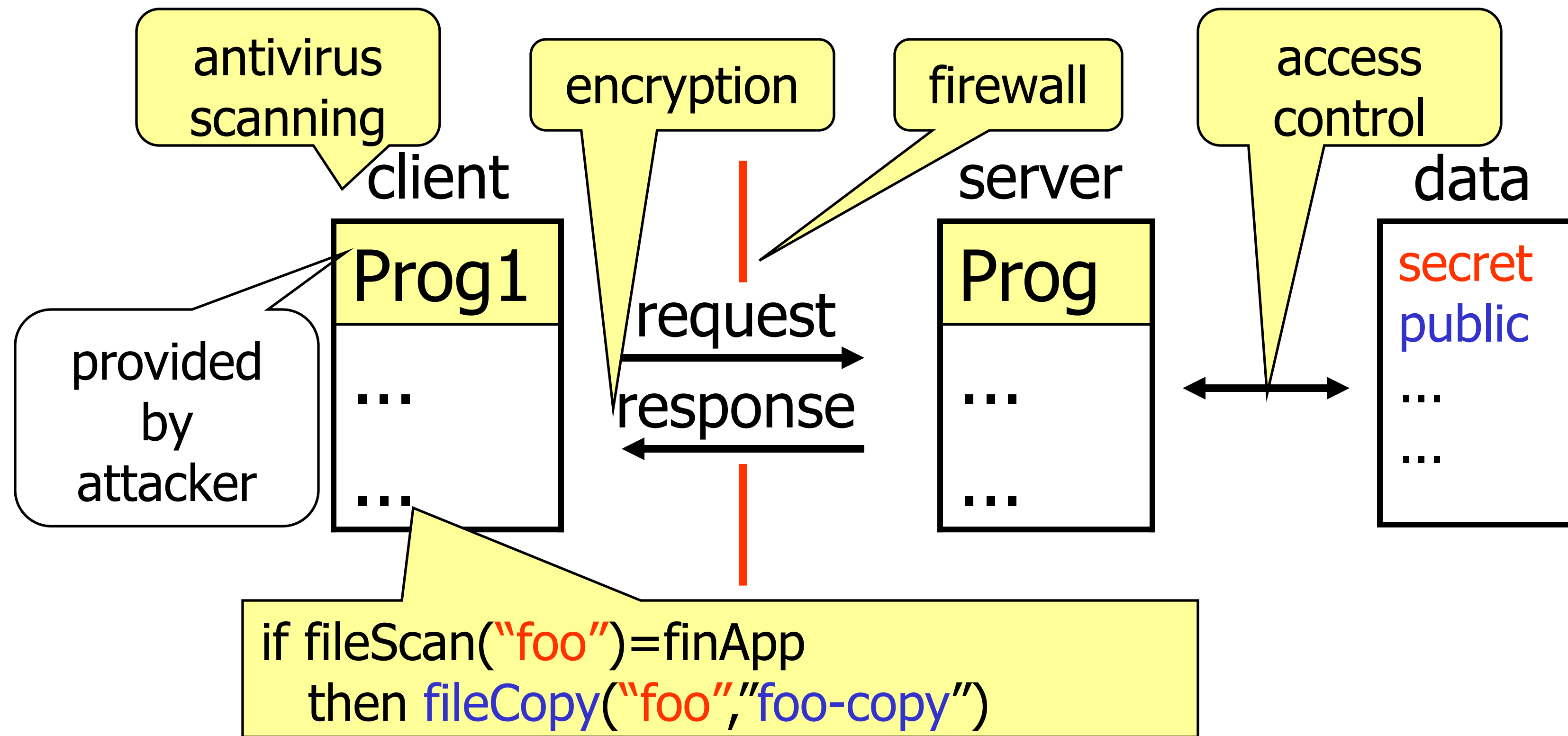


Computer Security

- The CIA
  - Confidentiality
  - Integrity
  - Availability



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

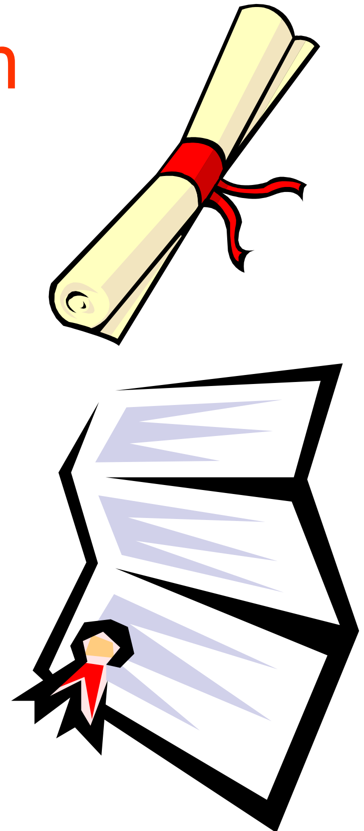
+good for low-level events (such as read a file)

-programs treated as black boxes

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

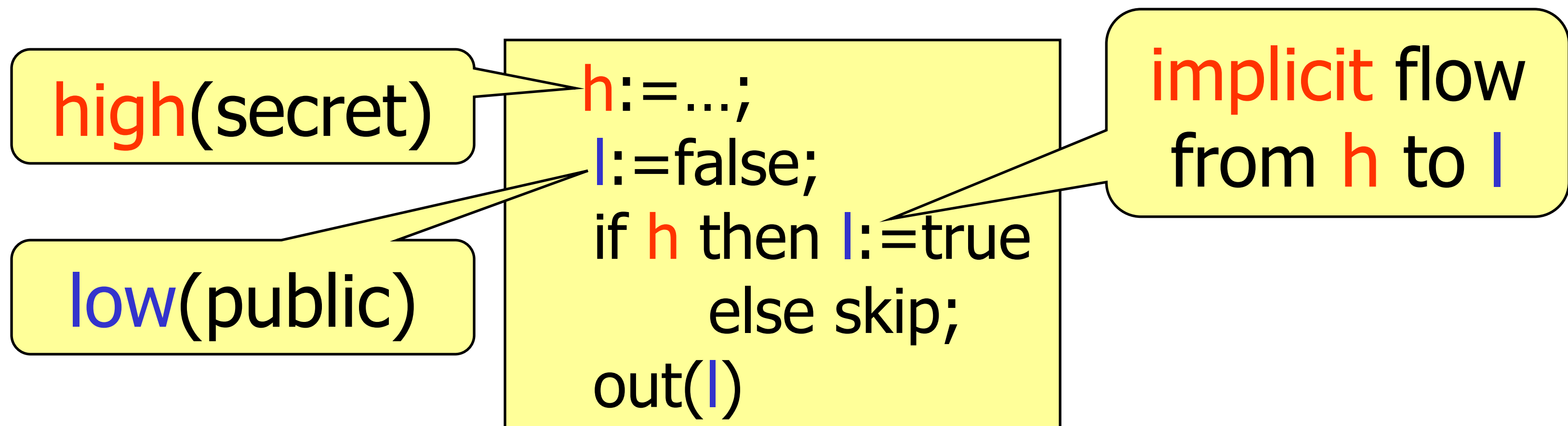
Confidentiality: language-based approach

- 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
- 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: 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

A security-type system

Expressions:

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

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

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

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

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

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] $\vdash$ if h then h:=h+7 else skip

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

[pc] $\nvdash$ while h<4 do l:=l+1

Type Inference: Example

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

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]:

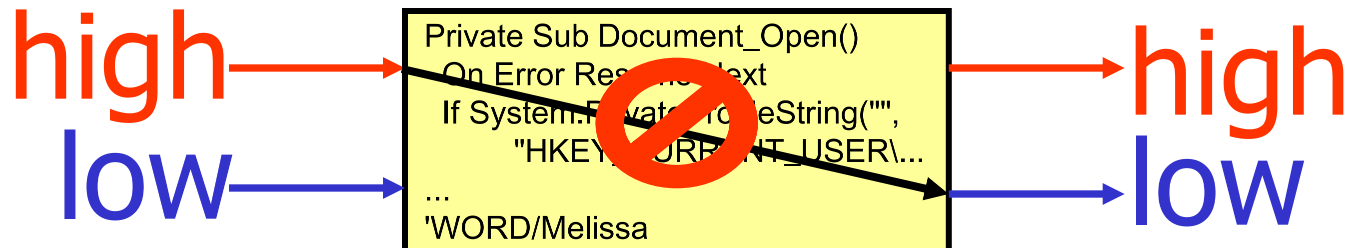
> A program is secure iff high inputs do not interfere with low-level view of the system

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:

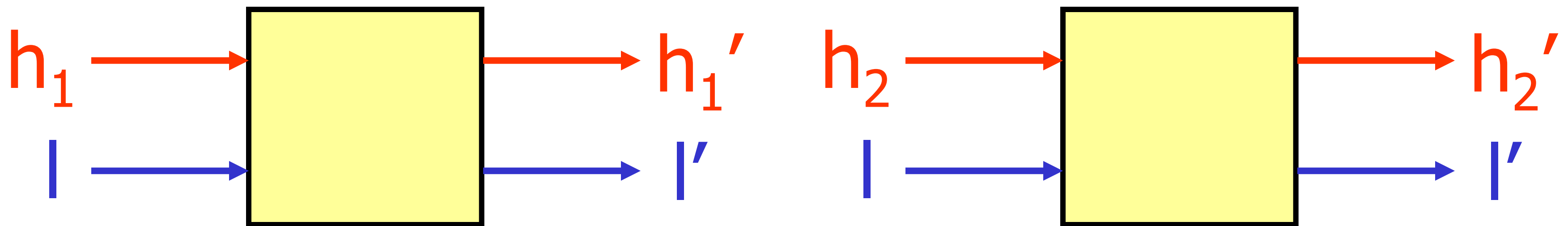
secret (high)

public (low)

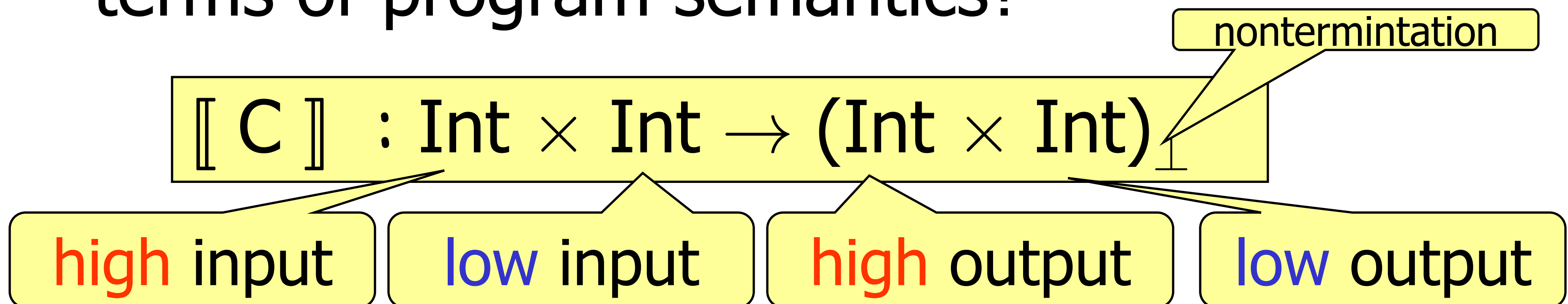


Confidentiality for sequential programs: noninterference

- 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:

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

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

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

Low view $\approx_L$: indistinguishability by attacker

C is secure iff

$$\forall mem_1, mem_2.\ mem_1 =_L mem_2 \Rightarrow [\![\ C\ ]\!]mem_1 \approx_L [\![\ C\ ]\!]mem_2$$

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 $\approx_L$:

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

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

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

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

Static certification

Noninterference

Procedures

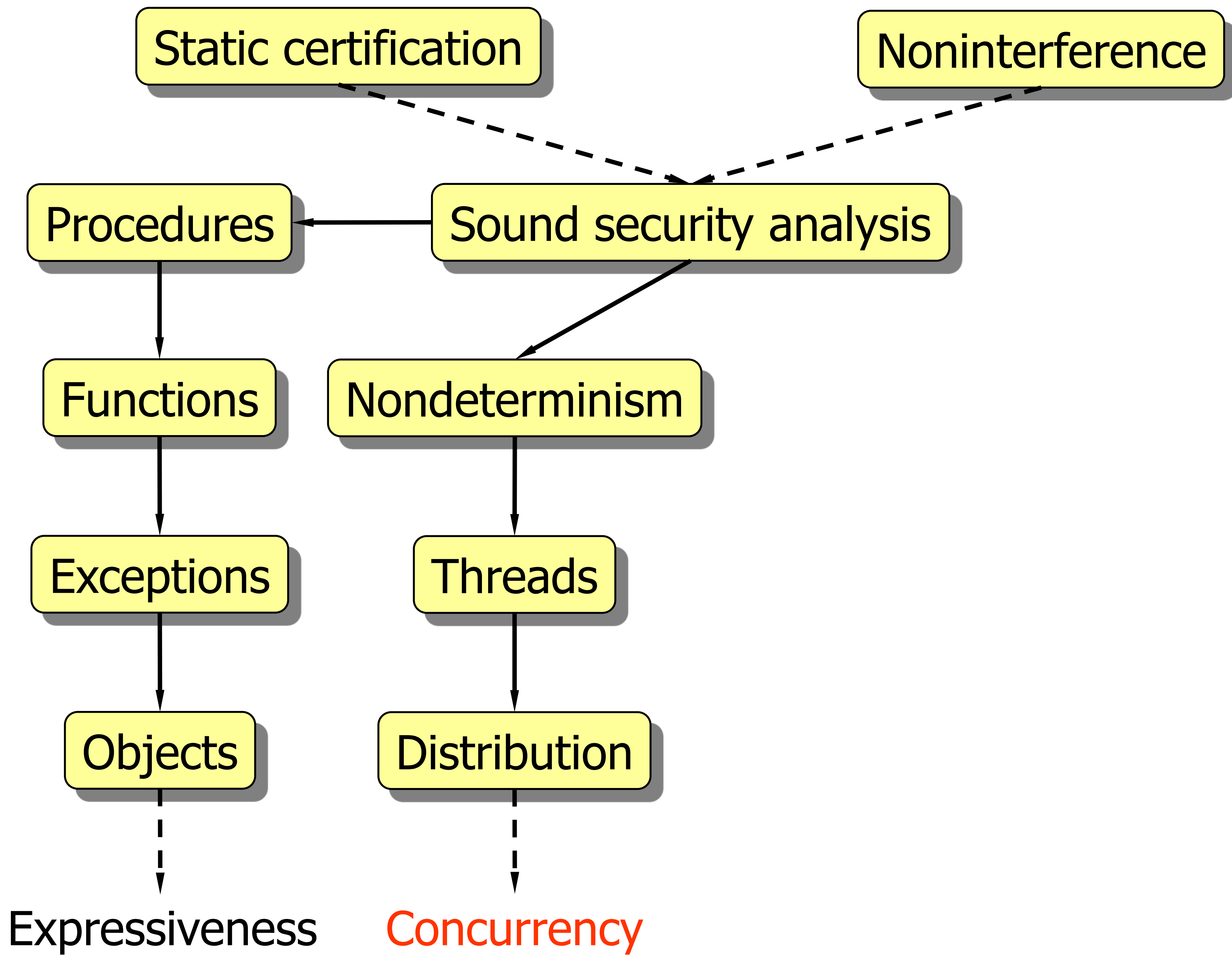
Sound security analysis

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

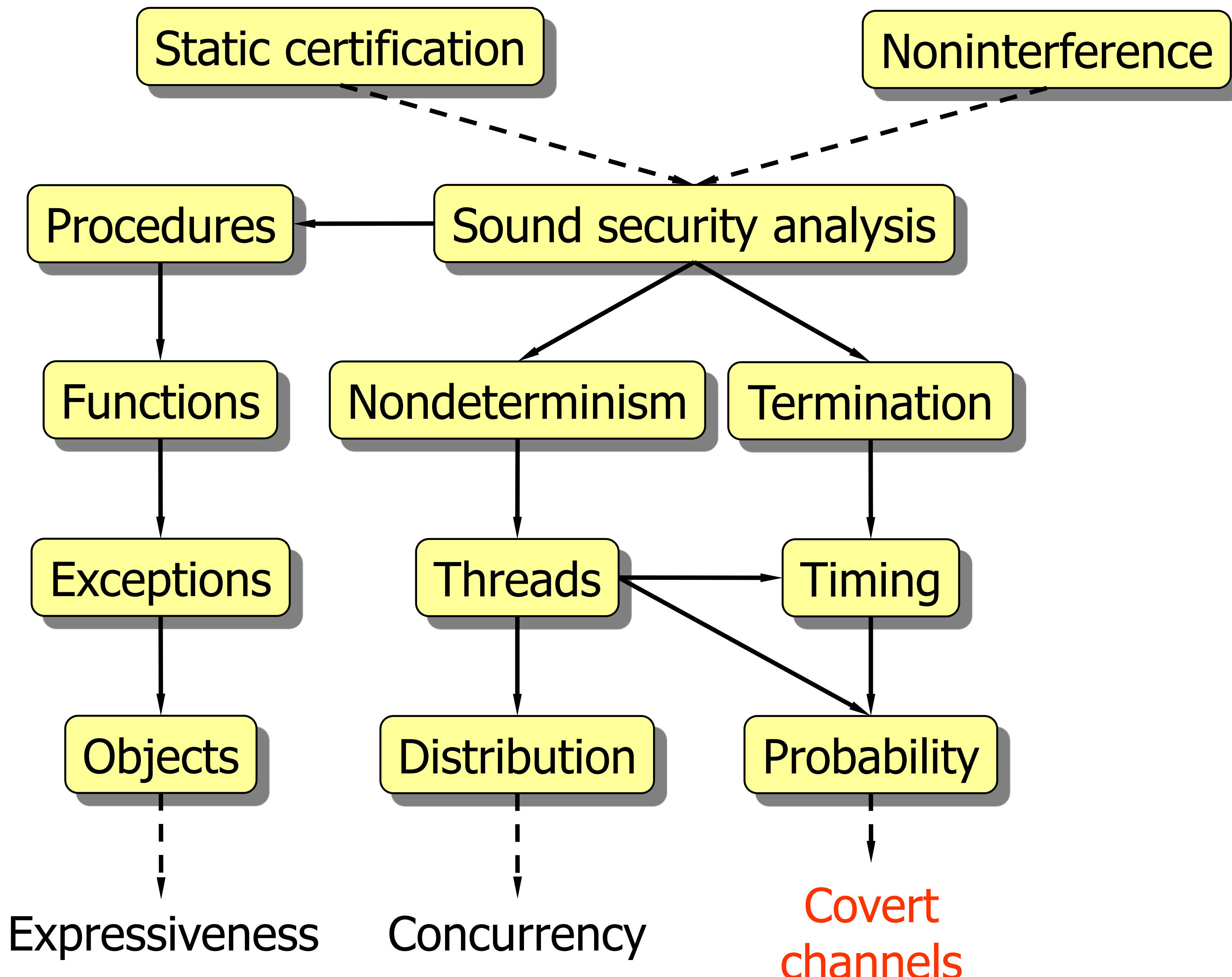
- Blocking a process: observable by other processes (also timing, probabilities,...)

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

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]



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):

$$\text{mem} \approx_L \text{mem}' \text{ iff}$$
$$\text{mem} = \bot = \text{mem}' \lor$$
$$(\text{mem} \neq \bot \neq \text{mem}' \land \text{mem} =_L \text{mem}')$$

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

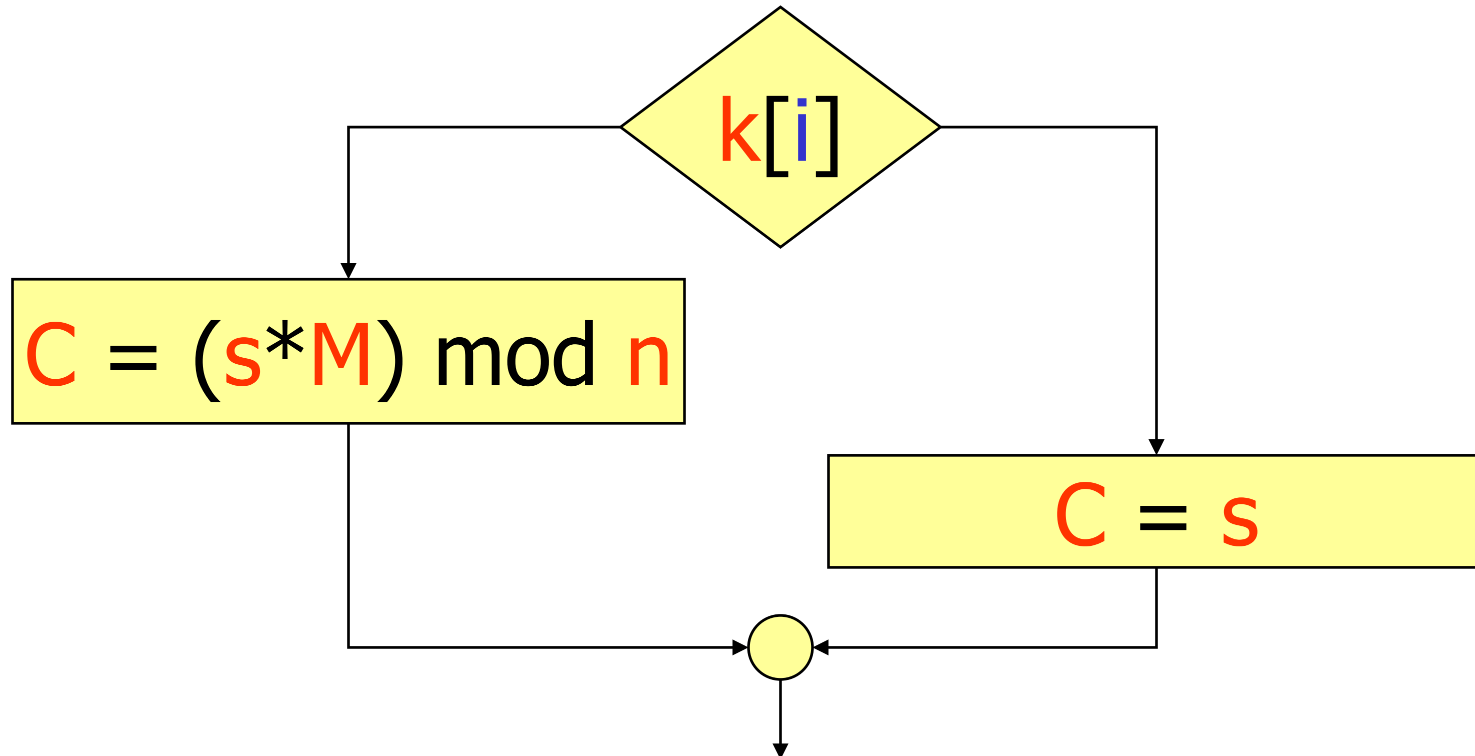
```
s = 1;
for (i=0; i<w; i++){
  if (k[i])
     C = (s*M) mod n;
  else
     C = s;
  s = C*C;
}
```

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

[Kocher'96]

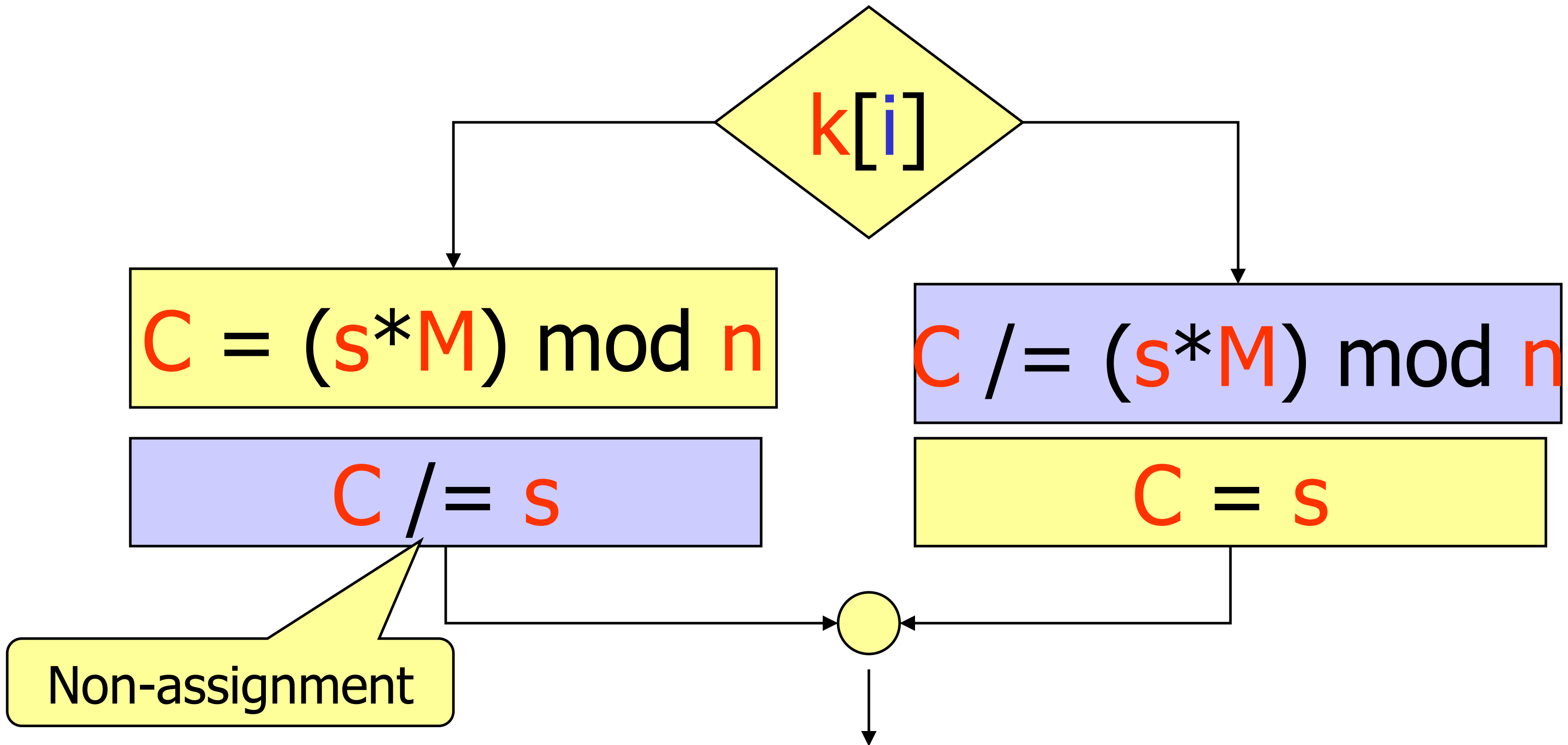
Transforming out timing leaks

Branching on high causes leaks



Transforming out timing leaks

Cross-copy low slices



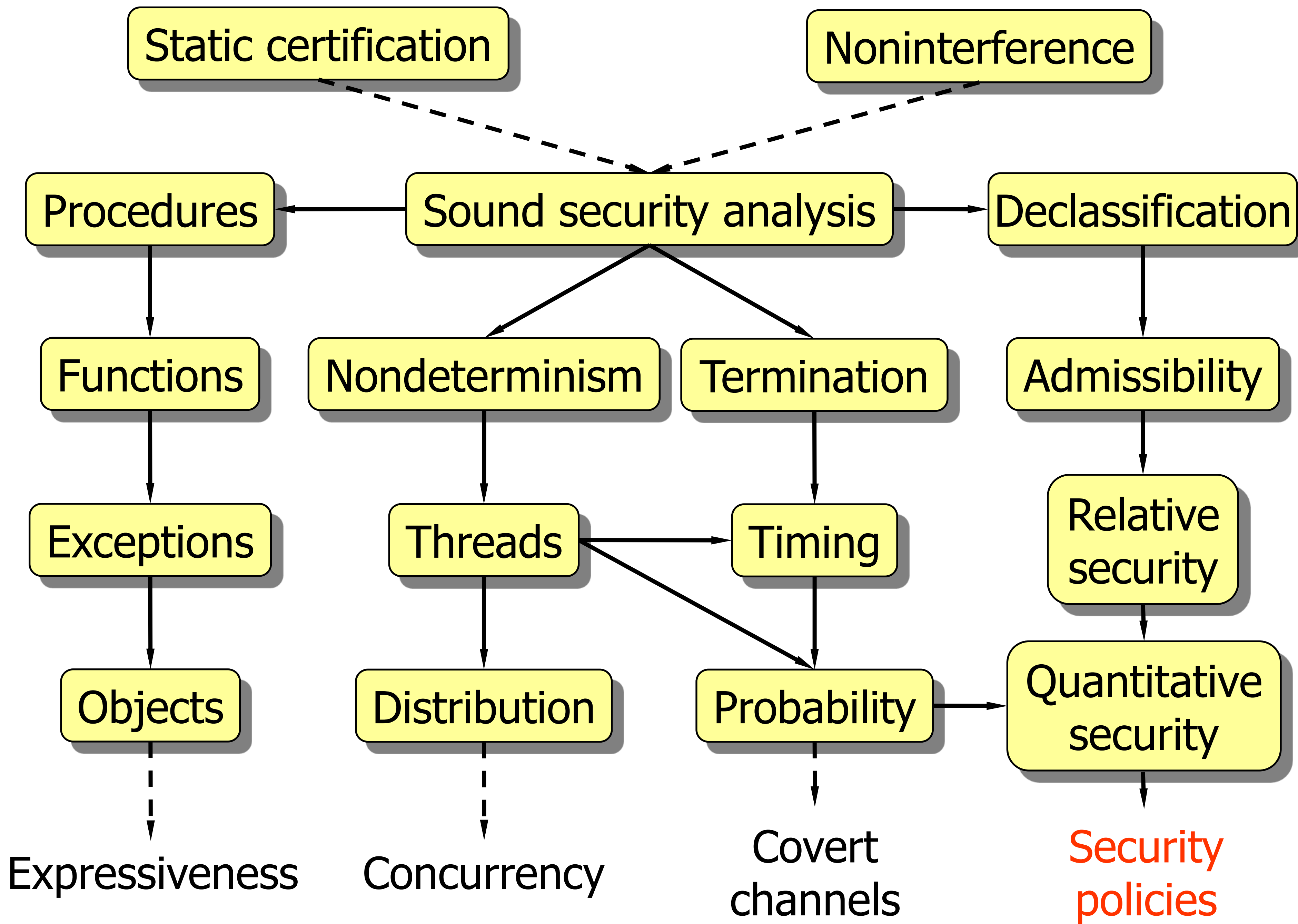
Covert channels: Probabilistic

- Possibilistically but not probabilistically secure program: $l := PIN \; \square_{9/10} \; l := rand(9999)$
- 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]


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 in the next lecture

Most 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 in the next lecture

More on this in the next lecture

More on this in the next lecture

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 Information-Flow Security: the big picture
3. Dimensions and principles of declassification
4. Combining the dimensions of declassification for dynamic languages

tomor-row (items 3–4)

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]

End of talk